**ЛАБОРАТОРНАЯ РАБОТА №8. ИНТЕРФЕЙСЫ**

**Цель**: Научиться создавать и реализовывать интерфейсы. Освоить создание программы, состоящей из нескольких пакетов.

**Краткие теоретические сведения.**

**Интерфейсы**

Механизм наследования очень удобен, но он имеет свои ограничения. В частности, мы можем наследовать только от одного класса, в отличие, например, от языка С++, где имеется множественное наследование.

В языке C# подобную проблему частично позволяют решить интерфейсы. Интерфейсы определяют некоторый функционал,   
не имеющий конкретной реализации, который затем реализуют классы, применяющие эти интерфейсы. Один класс может применить множество интерфейсов.

Чтобы определить интерфейс, используется ключевое слово **interface**. Например:

interface Printable **{**

void print**();**

**}**

Данный интерфейс называется *Printable*. Интерфейс может определять константы и методы, которые могут иметь, а могут и не иметь реализации. Методы без реализации похожи на абстрактные методы абстрактных классов. Так, в данном случае объявлен один метод, который не имеет реализации.

Все методы интерфейса не имеют модификаторов доступа,   
но фактически по умолчанию у них доступ **public**, так как цель интерфейса – определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Чтобы класс применил интерфейс, надо использовать : как при наследовании:

interface Printable **{**

void print**();**

**}**

class Book: Printable **{**

private String name**;**

private String author**;**

public Book**(**String name**,** String author**)** **{**

**this.**name **=** name**;**

**this.**author **=** author**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public String getAuthor**()** **{**

**return** author**;**

**}**

public void setAuthor**(**String author**)** **{**

**this.**author **=** author**;**

**}**

public void print**()** **{**

Console.WriteLine**(**"{0} ({1})"**,** name**,** author**);**

**}**

**}**

В данном случае класс *Book* реализует интерфейс *Printable*. При этом надо учитывать, что если класс применяет интерфейс,   
то он должен реализовать все методы интерфейса, как в случае выше реализован метод *print*. Потом в методе *main* мы можем создать объект класса *Book* и вызвать его метод *print*.

Book book **=** **new** Book**(**"Name"**,** "Author"**);**

book**.**print**();**

Name (Author)

Если класс не реализует какие-то методы интерфейса, то такой класс должен быть определен как абстрактный, а его неабстрактные классы-наследники затем должны будут эти методы реализовать. В тоже время мы не можем напрямую создавать объекты интерфейсов, поэтому следующий код не будет работать:

Printable pr **=** **new** Printable**();**

pr**.**print**();**

Одним из преимуществ использования интерфейсов является то, что они позволяют добавить в приложение гибкости. Например,   
в дополнение к классу *Book* определим еще один класс, который будет реализовывать интерфейс *Printable*:

class Journal: Printable **{**

private String name**;**

public Journal**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public void print**()** **{**

Console.WriteLine**(**name**);**

**}**

**}**

Класс *Book* и класс *Journal* связаны тем, что они реализуют интерфейс *Printable*. Поэтому мы динамически в программе можем создавать объекты *Printable* как экземпляры обоих классов:

Printable printable **=** **new** Book**(**"Книга"**,** "Автор"**);**

printable**.**print**();**

printable **=** **new** Journal**(**"Журнал"**);**

printable**.**print**();**

Книга (Автор)

Журнал

**Интерфейсы в преобразованиях типов**

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Например, так как класс *Journal* реализует интерфейс *Printable*, то переменная типа *Printable* может хранить ссылку на объект типа *Journal*:

Printable p **=** **new** Journal**(**"Foreign Policy"**);**

p**.**print**();**

String name **=** **((**Journal**)**p**).**getName**();**

Console.WriteLine**(**name**);**

И если мы хотим обратиться к методам класса *Journal*, которые определены не в интерфейсе *Printable*, а в самом классе *Journal*, то нам надо явным образом выполнить преобразование типов.

**Методы по умолчанию**

Ранее, при реализации интерфейса мы должны были обязательно реализовать все его методы в классе, а сам интерфейс мог содержать только определения методов без конкретной реализации. В последних версиях языка была добавлена такая функциональность как методы по умолчанию. И теперь интерфейсы кроме определения методов могут иметь их реализацию по умолчанию, которая используется, если класс, реализующий данный интерфейс, не реализует метод. Например, создадим метод по умолчанию в интерфейсе *Printable*:

interface Printable **{**

**public** void print**()** **{**

Console.WriteLine**(**"Undefined printable!"**);**

**}**

**}**

Затем в классе *Journal* нам необязательно этот метод реализовать, хотя мы можем его и переопределить:

class Journal: Printable **{**

private String name**;**

public Journal**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

**}**

Printable p **=** **new** Journal**(**"Foreign Policy"**);**

p**.**print**();**

Undefined printable!

**Множественная реализация интерфейсов**

Если нам надо применить в классе несколько интерфейсов,   
то они все перечисляются через запятую:

interface Printable **{**

void print**();**

**}**

interface Searchable **{**

void search**();**

**}**

class Book: Printable**,** Searchable **{**

public void print**()** **{**

**...**

**}**

public void search**()** **{**

**...**

**}**

**}**

**Наследование интерфейсов**

Интерфейсы, как и классы, могут наследоваться:

interface Printable **{**

void print**();**

**}**

interface Searchable **{**

void search**();**

**}**

interface BookStuff: Printable**,** Searchable **{**

void reference**();**

**}**

class Book: BookStuff **{**

public void print**()** **{**

**...**

**}**

public void search**()** **{**

**...**

**}**

public void reference**()** **{**

**...**

**}**

**}**

При применении этого интерфейса класс *Book* должен будет реализовать как методы интерфейса *BookStuff*, так и методы базовых интерфейсов *Printable* и *Searchable*.

**Вложенные интерфейсы**

Как и классы, интерфейсы могут быть вложенными, то есть могут быть определены в классах или других интерфейсах. Например:

class Printer **{**

interface Printable **{**

void print**();**

**}**

**}**

Использование интерфейса будет аналогично предыдущим случаям:

Printer**.**Printable p **=** **new** Journal**(**"Foreign Affairs"**);**

**Интерфейсы как параметры и результаты методов**

Так же, как и в случае с классами, интерфейсы могут использоваться в качестве типа параметров метода или в качестве возвращаемого типа:

interface Printable **{**

void print**();**

**}**

class Book: Printable **{**

private String name**;**

private String author**;**

public Book**(**String name**,** String author**)** **{**

**this.**name **=** name**;**

**this.**author **=** author**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public String getAuthor**()** **{**

**return** author**;**

**}**

public void setAuthor**(**String author**)** **{**

**this.**author **=** author**;**

**}**

public void print**()** **{**

Console.WriteLine**(**"{0} ({1})"**,** name**,** author**);**

**}**

**}**

class Journal: Printable **{**

private String name**;**

public Journal**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public String getName**()** **{**

**return** name**;**

**}**

public void setName**(**String name**)** **{**

**this.**name **=** name**;**

**}**

public void print**()** **{**

Console.WriteLine**(**name**);**

**}**

**}**

public class Program **{**

public static void read**(**Printable p**)** **{**

p**.**print**();**

**}**

public static Printable createPrintable**(**bool option**,** String name**)** **{**

**if** **(**option**)** **{**

**return** **new** Book**(**name**,** "Undefined"**);**

**}** **else** **{**

**return** **new** Journal**(**name**);**

**}**

**}**

public static void main**(**String**...**args**)** **{**

Printable p **=** createPrintable**(false,** "Foreign Affairs"**);**

p**.**print**();**

read**(new** Book**(**"Книга"**,** "Автор"**));**

read**(new** Journal**(**"Журнал"**));**

**}**

**}**

Foreign Affairs

Книга (Автор)

Журнал

Метод *read()* в качестве параметра принимает объект интерфейса *Printable*, поэтому в этот метод мы можем передать как объект *Book*, так и объект *Journal*. Метод *createPrintable()* возвращает объект *Printable*, поэтому мы также можем возвратить как объект *Book*, так и *Journal*.

**Интерфейсы в механизме обратного вызова**

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова. Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях. Стандартный пример – нажатие на кнопку. Когда мы нажимаем на кнопку, мы производим действие, но в ответ на это нажатие запускаются другие действия. Например, нажатие   
на значок принтера запускает печать документа на принтере и т.д.

interface EventHandler **{**

void execute**();**

**}**

class ButtonClickHandler: EventHandler **{**

public void execute**()** **{**

Console.WriteLine**(**"Button is pressed!"**);**

**}**

**}**

class Button **{**

private EventHandler handler**;**

public Button**(**EventHandler action**)** **{**

**this.**handler **=** action**;**

**}**

public void click**()** **{**

handler**.**execute**();**

**}**

**}**

Button b **=** **new** Button**(new** ButtonClickHandler**());**

b**.**click**();**

b**.**click**();**

Button is pressed!

Button is pressed!

Итак, здесь у нас определен класс *Button*, который в конструкторе принимает объект интерфейса *EventHandler* и в методе *click* (имитация нажатия) вызывает метод *execute* этого объекта.

Далее определяется реализация *EventHandler* в виде класса *ButtonClickHandler,* и в основной программе объект этого класса передается в конструктор Button. Таким образом, через конструктор мы устанавливаем обработчик нажатия кнопки, и при каждом вызове метода *button.click()* будет вызываться этот обработчик.

Но, казалось бы, зачем нам выносить все действия в интерфейс, его реализовывать. Почему бы не написать проще сразу   
в классе *Button*:

class Button **{**

public void click**()** **{**

Console.WriteLine**(**"Button is pressed!"**);**

**}**

**}**

Дело в том, что на момент определения класса нам не всегда бывают точно известны те действия, которые должны производиться. Особенно если класс *Button* и класс основной программы находятся в разных библиотеках и могут проектироваться разными разработчиками. К тому же у нас может быть несколько кнопок – объектов *Button*, и для каждого объекта надо определить свое действие.

**Порядок выполнения работы**

В лабораторную работу № 2 нужно внести следующие изменения:

* добавить возможность сравнивать объекты между собой   
  по умолчанию и по любому выбранному полю базового класса.

Для демонстрации нужно разработать программу,   
в которой:

* для хранения объектов используется список;
* в список добавляется 15 объектов (объекты создаются с помощью конструкторов с параметрами);
* осуществляется вывод объектов списка на консоль;
* осуществляется сортировка и повторный вывод объектов списка на консоль.

За сравнение объектов по умолчанию отвечает интерфейс *[BaseClass]Comparable*. Этот интерфейс содержит метод *compareTo,* который в качестве аргумента принимает ссылку на объект базового класса, а возвращает целочисленное значение – результат сравнения текущего объекта   
и объекта, переданного в аргументе метода (< 0 – меньше, == 0 – равно, > 0 – больше).

Базовый класс должен реализовывать интерфейс *[BaseClass]Comparable*. Базовый класс может реализовать интерфейс,   
а дочерние классы должны это сделать таким образом, чтобы массив или список объектов можно было упорядочить сначала по классам, а затем по любому из атрибутов дочернего класса.

Дополнительный компоратор по полям базового класса реализуется с помощью интерфейса *[BaseClass]Comparator*. Этот интерфейс содержит метод *compare,* который в качестве аргументов принимает ссылки на два объекта базового класса, а возвращает целочисленное значение – результат их сравнения (< 0 – меньше, == 0 – равно, > 0 – больше).

Классы *[BaseClass]Comparator\** реализуют интерфейс *[BaseClass] Comparator* – реализация метода *compare* определяет порядок элементов.

Класс список *[BaseClass]List* может быть реализован на основе массива, ссылок или быть наследником от *List<[BaseClass]>*.   
У класса списка должны быть следующие методы:

* добавление элемента в конец списка;
* добавление элемента в произвольное положение в списке;
* удаление элемента по индексу;
* удаление всех элементов из списка;
* получение элемента по индексу;
* изменение элемента по индексу;
* сортировка списка по порядку, предусмотренному по умолчанию;
* сортировка списка с помощью выбранного компаратора;
* вывод списка на консоль.

**Контрольные вопросы**

1. Что такое интерфейсы? Чем интерфейсы отличаются   
   от классов?
2. Как осуществляется описание интерфейса?
3. Как осуществляется описание членов интерфейса?
4. Как осуществляется реализация интерфейса?
5. Что нужно сделать для того, чтобы в классе можно было реализовать больше одного интерфейса?
6. Как один интерфейс может наследовать другой интерфейс? Может ли он в этом случае наследовать больше одного интерфейса?

**Содержание отчета**

1. Ф.И.О., группа, номер, название лабораторной работы.
2. Цель работы.
3. Использованные в работе теоретические сведения.
4. Определения и объяснения, используемых интерфейсов.
5. Результат работы программы.
6. Листинг программы.
7. Выводы.

**Порядок защиты лабораторной работы**

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Ответ на вопросы по теме текущей лабораторной работы.

Дополнительная информация: <https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.sort?view=net-5.0>