ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Утверждаю

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
учебной-практической работы №5

учебной практики   
*УП 05. Программирование*

*Тема: «Интерфейсы»*

г. Томск – 2020 г

РАССМОТРЕННО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБОУ СПО «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Сидиков И.Д.

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Владимировна А. В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика учебной-практической работы №5 4](#_Toc22557890)

[2. Краткие теоретические сведения 5](#_Toc22557891)

[3. Задания для выполнения 45](#_Toc22557892)

[4. Индивидуальные задания 46](#_Toc22557893)

[5. Контрольные вопросы 47](#_Toc22557894)

[6. Рекомендованная литература 48](#_Toc22557895)

[ПРИЛОЖЕНИЕ А 49](#_Toc22557896)

[ПРИЛОЖЕНИЕ Б 50](#_Toc22557897)

# ОБЩАЯ ХАРАКТЕРИСТИКА УЧЕБНОЙ-ПРАКТИЧЕСКОЙ РАБОТЫ №5

**Тема:** «Интерфейсы».

**Цель работы:** получение первоначальных навыков по работе с интерфейсами на языке программирования C#

**Проверяемые компетенции:**

ОК 1, ОК 2, ОК 4, ОК 9, ПК11.2.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы.
2. Выполните предложенное практическое задание.
3. Оформите отчет по учебной практике (шаблон представлен в приложении)
4. Отправьте отчет на проверку.

**Время выполнения заданий:** 6 часов.

**Критерии оценки:**

|  |  |  |
| --- | --- | --- |
| **№** | **Критерий** | **Баллы** |
| 1 | Отчет соответствует предложенному шаблону | 1 |
| 2 | Задание выполнено правильно | 2 |
| 3 | В коде присутствует функции | 4 |
| 4 | Составлена блок схема к программе | 2 |
| 5 | Программа работает правильно без сбоев | 2 |
| 6 | Студент ответил на контрольные вопросы по лабораторной работе | 2 |
| 7 | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 1 |
| 8 | Своевременность выполнения задания | 1 |
| Итого | | 15 |

**Перевод в пятибалльную систему оценивания:**

«отлично» - 13 - 15 баллов

«хорошо» - 10 -12 баллов

«удовлетворительно» - 7-9

«неудовлетворительно» <7 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## Введение интерфейсы

Интерфейс содержит определения для группы связанных функциональных возможностей, которые может реализовать неабстрактный класс или структура. Интерфейс может определять методы static, которые должны иметь реализацию. Интерфейс может предоставлять реализацию по умолчанию для любого или для всех своих объявленных членов экземпляра. Интерфейс не может объявлять данные экземпляра, такие как поля, автоматические реализуемые свойства или события, подобные свойствам.

С помощью интерфейсов можно, например, включить в класс поведение из нескольких источников. Эта возможность очень важна в C#, поскольку этот язык не поддерживает множественное наследование классов. Кроме того, необходимо использовать интерфейс, если требуется имитировать наследование для структур, поскольку они не могут фактически наследовать от другой структуры или класса.

Интерфейс определяется с помощью ключевого слова interface, как показано в следующем примере.

|  |
| --- |
| С# |
| interface IEquatable<T>  {  bool Equals(T obj);  } |

Имя интерфейса должно быть допустимым именем идентификатора C#. По соглашению имена интерфейсов начинаются с заглавной буквы I.

Любой объект (класс или структура), реализующий интерфейс IEquatable<T>, должен содержать определение для метода Equals, соответствующее сигнатуре, которую задает интерфейс. В результате вы можете быть уверены, что класс, реализующий IEquatable<T>, содержит метод Equals, с помощью которого экземпляр этого класса может определить, равен ли он другому экземпляру того же класса.

Определение IEquatable<T> не предоставляет реализацию для метода Equals. Класс или структура может реализовывать несколько интерфейсов, но класс может наследовать только от одного класса.

Интерфейсы могут содержать методы экземпляра, свойства, события, индексаторы, а также любое сочетание этих четырех типов членов. Интерфейсы могут содержать статические конструкторы, поля, константы или операторы. Интерфейс не может содержать поля экземпляров, конструкторы экземпляров или методы завершения. Члены интерфейса по умолчанию являются открытыми.

Для реализации члена интерфейса соответствующий член реализующего класса должен быть открытым и не статическим, а также иметь такое же имя и сигнатуру, что и член интерфейса.

Если класс (или структура) реализует интерфейс, этот класс (или структура) должен предоставлять реализацию для всех членов, которые этот интерфейс объявляет, но не предоставляет им реализацию по умолчанию. Однако если базовый класс реализует интерфейс, то любой класс, производный от базового класса, наследует эту реализацию.

В следующем примере показана реализация интерфейса IEquatable<T>. Реализующий класс Car должен предоставлять реализацию метода Equals.

|  |
| --- |
| С# |
| public class Car : IEquatable<Car>  {  public string Make {get; set;}  public string Model { get; set; }  public string Year { get; set; }  // Implementation of IEquatable<T> interface  public bool Equals(Car car)  {  return (this.Make, this.Model, this.Year) ==  (car.Make, car.Model, car.Year);  }  } |

Свойства и индексаторы класса могут определять дополнительные методы доступа для свойства или индексатора, определенного в интерфейсе. Например, интерфейс может объявлять свойство, имеющее акцессор get. Класс, реализующий этот интерфейс, может объявлять это же свойство с обоими акцессорами (get и set). Однако если свойство или индексатор использует явную реализацию, методы доступа должны совпадать.

Интерфейс может наследовать от одного или нескольких интерфейсов. Производный интерфейс наследует члены от своих базовых интерфейсов. Класс, реализующий производный интерфейс, должен реализовывать все члены в нем, включая все члены базовых интерфейсов производного интерфейса. Этот класс может быть неявно преобразован в производный интерфейс или любой из его базовых интерфейсов. Класс может включать интерфейс несколько раз через наследуемые базовые классы или через интерфейсы, которые наследуются другими интерфейсами. Однако класс может предоставить реализацию интерфейса только однократно и только если класс объявляет интерфейс как часть определения класса (class ClassName : InterfaceName). Если интерфейс наследуется, поскольку наследуется базовый класс, реализующий этот интерфейс, то базовый класс предоставляет реализацию членов этого интерфейса. Но производный класс может повторно реализовать любые члены виртуального интерфейса и не использовать наследованную реализацию. Когда интерфейсы объявляют реализацию метода по умолчанию, любой класс, реализующий данный интерфейс, наследует эту реализацию. Реализации, определенные в интерфейсах, являются виртуальными, и реализующий класс может переопределить эту реализацию.

Базовый класс также может реализовывать члены интерфейса с помощью виртуальных членов. В таком случае производный класс может изменять поведение интерфейса путем переопределения виртуальных членов.

## Сводка по интерфейсам

Интерфейс имеет следующие свойства:

* Интерфейс обычно подобен абстрактному базовому классу, имеющему только абстрактные члены. Любой класс (или структура), реализующий интерфейс, должен реализовывать все его члены. При необходимости интерфейс может определять реализации по умолчанию для некоторых или для всех его членов.
* Невозможно создать экземпляр интерфейса напрямую. Его члены реализуются любым классом (или структурой), реализующим интерфейс.
* Класс или структура может реализовывать несколько интерфейсов. Класс может наследовать базовому классу и также реализовывать один или несколько интерфейсов.

## Определение интерфейса

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

* Методы
* Свойства
* Индексаторы
* События
* Статические поля и константы (начиная с версии C# 8.0)

Однако интерфейсы не могут определять нестатические переменные. Например, простейший интерфейс, который определяет все эти компоненты:

|  |
| --- |
| С# |
| interface IMovable  {      // константа      const int minSpeed = 0;     // минимальная скорость      // статическая переменная      static int maxSpeed = 60;   // максимальная скорость      // метод      void Move();                // движение      // свойство      string Name { get; set; }   // название        delegate void MoveHandler(string message);  // определение делегата для события      // событие      event MoveHandler MoveEvent;    // событие движения  } |

В данном случае определен интерфейс IMovable, который представляет некоторый движущийся объект. Данный интерфейс содержит различные компоненты, которые описывают возможности движущегося объекта. То есть интерфейс описывает некоторый функционал, который должен быть у движущегося объекта.

Методы и свойства интерфейса могут не иметь реализации, в этом они сближаются с абстрактными методами и свойствами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не имеет реализации, не принимает никаких параметров и ничего не возвращает.

То же самое в данном случае касается свойства Name. На первый взгляд оно похоже на автоматическое свойство. Но в реальности это определение свойства в интерфейсе, которое не имеет реализации, а не автосвойство.

Еще один момент в объявлении интерфейса: если его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Это касается также и констант и статических переменных, которые в классах и структурах по умолчанию имееют модификатор private. В интерфейсах же они имеют по умолчанию модификатор public. И например, мы могли бы обратиться к константе minSpeed и переменной maxSpeed интерфейса IMovable:

|  |
| --- |
| С# |
| static void Main(string[] args)  {      Console.WriteLine(IMovable.maxSpeed);      Console.WriteLine(IMovable.minSpeed);  } |

Но также, начиная с версии C# 8.0, мы можем явно указывать модификаторы доступа у компонентов интерфейса:

|  |
| --- |
| С# |
| interface IMovable  {  public const int minSpeed = 0; // минимальная скорость  private static int maxSpeed = 60; // максимальная скорость  public void Move();  protected internal string Name { get; set; } // название  public delegate void MoveHandler(string message); // определение делегата для события  public event MoveHandler MoveEvent; // событие движения  } |

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Это значит, что мы можем определить в интерфейсах полноценные методы и свойства, которые имеют реализацию как в обычных классах и структурах. Например, определим реализацию метода Move по умолчанию:

|  |
| --- |
| С# |
| interface IMovable  {  // реализация метода по умолчанию  void Move()  {  Console.WriteLine("Walking");  }  } |

С реализацией свойств по умолчанию в интерфейсах дело обстоит несколько сложнее, поскольку мы не можем определять в интерфейсах нестатические переменные, соответственно в свойствах интерфейса мы не можем манипулировать состоянием объекта. Тем не менее реализацию по умолчанию для свойств мы тоже можем определять:

|  |
| --- |
| С# |
| interface IMovable  {      void Move()      {          Console.WriteLine("Walking");      }      // реализация свойства по умолчанию      // свойство только для чтения      int MaxSpeed { get { return 0; } }  } |

Стоит отметить, что если интерфейс имеет приватные методы и свойства (то есть с модификатором private), то они должны иметь реализацию по умолчанию. То же самое относится к любым статическим методам и свойствам (не обязательно приватным):

|  |
| --- |
| С# |
| interface IMovable  {  public const int minSpeed = 0; // минимальная скорость  private static int maxSpeed = 60; // максимальная скорость  // находим время, за которое надо пройти расстояние distance со скоростью speed  static double GetTime(double distance, double speed) => distance / speed;  static int MaxSpeed  {  get { return maxSpeed; }  set  {  if (value > 0) maxSpeed = value;  }  }  }  class Program  {  static void Main(string[] args)  {  Console.WriteLine(IMovable.MaxSpeed);  IMovable.MaxSpeed = 65;  Console.WriteLine(IMovable.MaxSpeed);  double time = IMovable.GetTime(100, 10);  Console.WriteLine(time);  }  } |

### Модификаторы доступа интерфейсов

Как и классы, интерфейсы по умолчанию имеют уровень доступа internal, то есть такой интерфейс доступен только в рамках текущего проекта. Но с помощью модификатора public мы можем сделать интерфейс общедоступным:

|  |
| --- |
| С# |
| public interface IMovable  {  void Move();  } |

Стоит отметить, что в Visual Studio есть специальный компонент для добавления нового интерфейса в отдельном файле. Для добавления интерфейса в проект можно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать Add-> New Item... и в диалоговом окне добавления нового компонента выбрать пункт Interface:

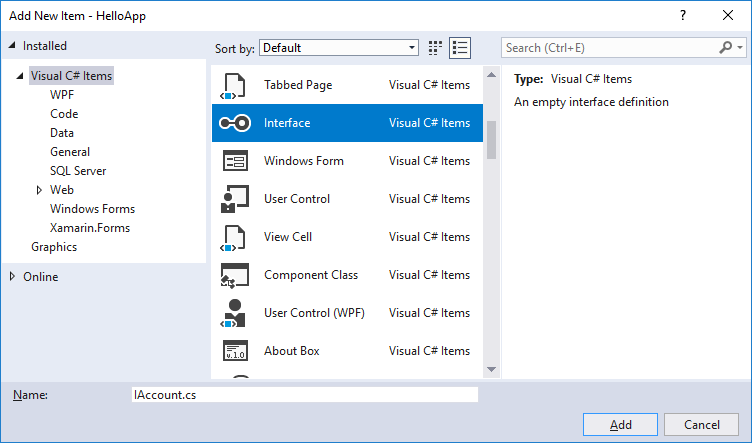


Рисунок 1. Создание интерфейса в Visual Studio

## Применение интерфейсов

Интерфейс представляет некое описание типа, набор компонентов, который должен иметь тип данных. И, собственно, мы не можем создавать объекты интерфейса напрямую с помощью конструктора, как например, в классах:

|  |
| --- |
| С# |
| IMovable m = new IMovable(); // ! Ошибка, так сделать нельзя |

В конечном счете интерфейс предназначен для реализации в классах и структурах. Например, возьмем следующий интерфейс IMovable:

|  |
| --- |
| С# |
| interface IMovable  {  void Move();  } |

Затем какой-нибудь класс или структура могут применить данный интерфейс:

|  |
| --- |
| С# |
| // применение интерфейса в классе  class Person : IMovable  {      public void Move()      {          Console.WriteLine("Человек идет");      }  }  // применение интерфейса в структуре  struct Car : IMovable  {      public void Move()      {          Console.WriteLine("Машина едет");      }  } |

При применении интерфейса, как и при наследовании после имени класса или структуры указывается двоеточие и затем идут названия применяемых интерфейсов. При этом класс должен реализовать все методы и свойства применяемых интерфейсов, если эти методы и свойства не имеют реализации по умолчанию.

Если методы и свойства интерфейса не имеют модификатора доступа, то по умолчанию они являются публичными, при реализации этих методов и свойств в классе и структуре к ним можно применять только модификатор public.

Применение интерфейса в программе:

|  |
| --- |
| С# |
| using System;  namespace HelloApp  {      interface IMovable      {          void Move();      }      class Person : IMovable      {          public void Move()          {              Console.WriteLine("Человек идет");          }      }      struct Car : IMovable      {          public void Move()          {              Console.WriteLine("Машина едет");          }      }      class Program      {          static void Action(IMovable movable)          {              movable.Move();          }          static void Main(string[] args)          {              Person person = new Person();              Car car = new Car();              Action(person);              Action(car);              Console.Read();          }      }  } |

В данной программе определен метод Action(), который в качестве параметра принимает объект интерфейса IMovable. На момент написания кода мы можем не знать, что это будет за объект - какой-то класс или структура. Единственное, в чем мы можем быть уверены, что этот объект обязательно реализует метод Move и мы можем вызвать этот метод.

Иными словами, интерфейс — это контракт, что какой-то определенный тип обязательно реализует некоторый функционал.

Консольный вывод данной программы:

|  |
| --- |
| С# |
| Человек идет  Машина едет |

### Реализация интерфейсов по умолчанию

Начиная с версии C# 8.0 интерфейсы поддерживают реализацию методов и свойств по умолчанию. Зачем это нужно? Допустим, у нас есть куча классов, которые реализуют некоторый интерфейс. Если мы добавим в этот интерфейс новый метод, то мы будем обязаны реализовать этот метод во всех классах, применяющих данный интерфейс. Иначе подобные классы просто не будут компилироваться. Теперь вместо реализации метода во всех классах нам достаточно определить его реализацию по умолчанию в интерфейсе. Если класс не реализует метод, будет применяться реализация по умолчанию.

|  |
| --- |
| С# |
| class Program  {  static void Main(string[] args)  {  IMovable tom = new Person();  Car tesla = new Car();  tom.Move(); // Walking  tesla.Move(); // Driving  }  }    interface IMovable  {  void Move()  {  Console.WriteLine("Walking");  }  }  class Person : IMovable { }  class Car : IMovable  {  public void Move()  {  Console.WriteLine("Driving");  }  } |

В данном случае интерфейс IMovable определяет реализацию по умолчанию для метода Move. Класс Person не реализует этот метод, поэтому он применяет реализацию по умолчанию в отличие от класса Car, который определяет свою реализацию для метода Move.

Стоит отметить, что хотя для объекта класса Person мы можем вызвать метод Move - ведь класс Person применяет интерфейс IMovable, тем не менее мы не можем написать так:

|  |
| --- |
| С# |
| Person tom = new Person();  tom.Move();     // Ошибка - метод Move не определен в классе Person |

### Множественная реализация интерфейсов

Интерфейсы имеют еще одну важную функцию: в C# не поддерживается множественное наследование, то есть мы можем унаследовать класс только от одного класса, в отличие, скажем, от языка С++, где множественное наследование можно использовать. Интерфейсы позволяют частично обойти это ограничение, поскольку в C# класс может реализовать сразу несколько интерфейсов. Все реализуемые интерфейсы указываются через запятую:

|  |
| --- |
| С# |
| myClass: myInterface1, myInterface2, myInterface3, ...  {  ///  } |

**Рассмотрим на примере:**

|  |
| --- |
| С# |
| using System;    namespace HelloApp  {  interface IAccount  {  int CurrentSum { get; } // Текущая сумма на счету  void Put(int sum); // Положить деньги на счет  void Withdraw(int sum); // Взять со счета  }  interface IClient  {  string Name { get; set; }  }  class Client : IAccount, IClient  {  int \_sum; // Переменная для хранения суммы  public string Name { get; set; }  public Client(string name, int sum)  {  Name = name;  \_sum = sum;  }    public int CurrentSum { get { return \_sum; } }    public void Put(int sum) { \_sum += sum; }    public void Withdraw(int sum)  {  if (\_sum >= sum)  {  \_sum -= sum;  }  }  }  class Program  {  static void Main(string[] args)  {  Client client = new Client("Tom", 200);  client.Put(30);  Console.WriteLine(client.CurrentSum); //230  client.Withdraw(100);  Console.WriteLine(client.CurrentSum); //130  Console.Read();  }  }  } |

В данном случае определены два интерфейса. Интерфейс IAccount определяет свойство CurrentSum для текущей суммы денег на счете и два метода Put и Withdraw для добавления денег на счет и изъятия денег. Интерфейс IClient определяет свойство для хранения имени клиента.

Обратите внимание, что свойства CurrentSum и Name в интерфейсах похожи на автосвойства, но это не автосвойства. При реализации мы можем развернуть их в полноценные свойства, либо же сделать автосвойствами.

Класс Client реализует оба интерфейса и затем применяется в программе.

### Интерфейсы в преобразованиях типов

Все сказанное в отношении преобразования типов характерно и для интерфейсов. Поскольку класс Client реализует интерфейс IAccount, то переменная типа IAccount может хранить ссылку на объект типа Client:

|  |
| --- |
| С# |
| // Все объекты Client являются объектами IAccount  IAccount account = new Client("Том", 200);  account.Put(200);  Console.WriteLine(account.CurrentSum); // 400  // Не все объекты IAccount являются объектами Client, необходимо явное приведение  Client client = (Client)account;  // Интерфейс IAccount не имеет свойства Name, необходимо явное приведение  string clientName = ((Client)account).Name; |

Преобразование от класса к его интерфейсу, как и преобразование от производного типа к базовому, выполняется автоматически. Так как любой объект Client реализует интерфейс IAccount.

Обратное преобразование - от интерфейса к реализующему его классу будет аналогично преобразованию от базового класса к производному. Так как не каждый объект IAccount является объектом Client (ведь интерфейс IAccount могут реализовать и другие классы), то для подобного преобразования необходима операция приведения типов. И если мы хотим обратиться к методам класса Client, которые не определены в интерфейсе IAccount, но являются частью класса Client, то нам надо явным образом выполнить преобразование типов: string clientName = ((Client)account).Name;

## Явная реализация интерфейсов

Кроме неявного применения интерфейсов, которое было рассмотрено в прошлой статье, сушествует также явная реализация интерфейса. При явной реализации указывается название метода или свойства вместе с названием интерфейса, при этом мы не можем использовать модификатор public, то есть методы являются закрытыми:

|  |
| --- |
| С# |
| interface IAction  {      void Move();  }  class BaseAction : IAction  {      void IAction.Move()      {          Console.WriteLine("Move in Base Class");      }  } |

Следует учитывать, что при явной реализации интерфейса его методы и свойства не являются частью интерфейса класса. Поэтому напрямую через объект класса мы к ним обратиться не сможем:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  BaseAction action = new BaseAction();  ((IAction)action).Move(); // необходимо приведение к типу IAction    // или так  IAction action2 = new BaseAction();  action2.Move();    Console.ReadKey();  } |

В какой ситуации может действительно понадобиться явная реализация интерфейса? Например, когда класс применяет несколько интерфейсов, но они имеют один и тот же метод с одним и тем же возвращаемым результатом и одним и тем же набором параметров:

|  |
| --- |
| С# |
| class Person : ISchool, IUniversity  {  public void Study()  {  Console.WriteLine("Учеба в школе или в университете");  }  }    interface ISchool  {  void Study();  }    interface IUniversity  {  void Study();  } |

Класс Person определяет один метод Study(), создавая одну общую реализацию для обоих примененных интерфейсов. И вне зависимости от того, будем ли мы рассматривать объект Person как объект типа ISchool или IUniversity, результат метода будет один и тот же.

Чтобы разграничить реализуемые интерфейсы, надо явным образом применить интерфейс:

|  |
| --- |
| С# |
| class Person : ISchool, IUniversity  {  void ISchool.Study()  {  Console.WriteLine("Учеба в школе");  }  void IUniversity.Study()  {  Console.WriteLine("Учеба в университете");  }  } |

Использование:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  Person p = new Person();    ((ISchool)p).Study();  ((IUniversity)p).Study();    Console.Read();  } |

Другая ситуация, когда в базовом классе уже реализован интерфейс, но необходимо в производном классе по-своему реализовать интерфейс:

|  |
| --- |
| С# |
| interface IAction  {      void Move();  }  class BaseAction : IAction  {      public void Move()      {          Console.WriteLine("Move in BaseAction");      }  }  class HeroAction : BaseAction, IAction  {      void IAction.Move()      {          Console.WriteLine("Move in HeroAction");      }  } |

Несмотря на то, что базовый класс BaseAction уже реализовал интерфейс IAction, но производный класс по-своему реализует его. Применение классов:

|  |
| --- |
| С# |
| HeroAction action1 = new HeroAction();  action1.Move();            // Move in BaseAction  ((IAction)action1).Move(); // Move in HeroAction    IAction action2 = new HeroAction();  action2.Move();             // Move in HeroAction |

### Модификаторы доступа

Члены интерфейса могут иметь разные модификаторы доступа. Если модификатор доступа не public, а какой-то другой, то для реализации метода, свойства или события интерфейса в классах и структурах также необходимо использовать явную реализацию интерфейса.

|  |
| --- |
| С# |
| interface IMovable  {  protected internal void Move();  protected internal string Name { get; set; }  delegate void MoveHandler();  protected internal event MoveHandler MoveEvent;  }  class Person : IMovable  {  // явная реализация свойства - в виде автосвойства  string IMovable.Name { get; set; }    // явная реализация события - дополнительно создается переменная  IMovable.MoveHandler \_moveEvent;  event IMovable.MoveHandler IMovable.MoveEvent  {  add => \_moveEvent += value;  remove => \_moveEvent -= value;  }    // явная реализация метода  void IMovable.Move()  {  Console.WriteLine("Person is walking");  \_moveEvent();  }  }  class Program  {  static void Main(string[] args)  {  IMovable mov = new Person();  // подписываемся на событие  mov.MoveEvent += () => Console.WriteLine("IMovable is moving");  mov.Move();  }  } |

В данном случае опять же надо учитывать, что напрямую мы можем обратиться к подобным методам, свойствам и событиям через переменную интерфейса, но не переменную класса.

## Реализация интерфейсов в базовых и производных классах

Если класс применяет интерфейс, то этот класс должен реализовать все методы и свойства интерфейса, которые не имеют реализации по умолчанию. Однако также можно и не реализовать методы, сделав их абстрактными, переложив право их реализации на производные классы:

|  |
| --- |
| С# |
| interface IMovable  {  void Move();  }  abstract class Person : IMovable  {  public abstract void Move();  }  class Driver : Person  {  public override void Move()  {  Console.WriteLine("Шофер ведет машину");  }  } |

При реализации интерфейса учитываются также методы и свойства, унаследованные от базового класса. Например:

|  |
| --- |
| С# |
| interface IAction  {  void Move();  }  class BaseAction  {  public void Move()  {  Console.WriteLine("Move in BaseAction");  }  }  class HeroAction : BaseAction, IAction  {  } |

Здесь класс HeroAction реализует интерфейс IAction, однако для реализации метода Move из интерфейса применяется метод Move, унаследованный от базового класса BaseAction. Таким образом, класс HeroAction может не реализовать метод Move, так как этот метод уже определен в базовом классе BaseAction.

Следует отметить, что если класс одновременно наследует другой класс и реализует интерфейс, как в примере выше класс HeroAction, то название базового класса должно быть указано до реализуемых интерфейсов: class HeroAction : BaseAction, IAction

### Изменение реализации интерфейсов в производных классах

Может сложиться ситуация, что базовый класс реализовал интерфейс, но в классе-наследнике необходимо изменить реализацию этого интерфейса. Что в этом случае делать? В этом случае мы можем использовать либо переопределение, либо сокрытие метода или свойства интерфейса.

Первый вариант - переопределение виртуальных/абстрактных методов:

|  |
| --- |
| С# |
| interface IAction  {  void Move();  }  class BaseAction : IAction  {  public virtual void Move()  {  Console.WriteLine("Move in BaseAction");  }  }  class HeroAction : BaseAction  {  public override void Move()  {  Console.WriteLine("Move in HeroAction");  }  } |

В базовом классе BaseAction реализованный метод интерфейса определен как виртуальный (можно было бы также сделать его абстрактным), а в производном классе он переопределен.

При вызове метода через переменную интерфейса, если она ссылается на объект производного класса, будет использоваться реализация из производного класса:

|  |
| --- |
| С# |
| BaseAction action1 = new HeroAction();  action1.Move(); // Move in HeroAction    IAction action2 = new HeroAction();  action2.Move(); // Move in HeroAction |

Второй вариант - сокрытие метода в производном классе:

|  |
| --- |
| С# |
| interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move()  {  Console.WriteLine("Move in BaseAction");  }  }  class HeroAction : BaseAction  {  public new void Move()  {  Console.WriteLine("Move in HeroAction");  }  } |

Также используем эти классы:

|  |
| --- |
| С# |
| BaseAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction    IAction action2 = new HeroAction();  action2.Move(); // Move in BaseAction |

Так как интерфейс реализован именно в классе BaseAction, то через переменную action2 можно обратиться только к реализации метода Move из базового класса BaseAction.

Третий вариант - повторная реализация интерфейса в классе-наследнике:

|  |
| --- |
| С# |
| interface IAction  {  void Move();  }  class BaseAction : IAction  {  public void Move()  {  Console.WriteLine("Move in BaseAction");  }  }  class HeroAction : BaseAction, IAction  {  public new void Move()  {  Console.WriteLine("Move in HeroAction");  }  } |

В этом случае реализации этого метода из базового класса будет игнорироваться:

|  |
| --- |
| С# |
| BaseAction action1 = new HeroAction();  action1.Move(); // Move in BaseAction    IAction action2 = new HeroAction();  action2.Move(); // Move in HeroAction |

Также стоит отметить, что в случае с переменной action1 по-прежнему действует ранее связывание, в силу которого через эту переменную можно вызвать реализацию метода Move только из базового класса, который эта переменная представляет.

## Наследование интерфейсов

Интерфейсы, как и классы, могут наследоваться:

|  |
| --- |
| С# |
| interface IAction  {  void Move();  }  interface IRunAction : IAction  {  void Run();  }  class BaseAction : IRunAction  {  public void Move()  {  Console.WriteLine("Move");  }  public void Run()  {  Console.WriteLine("Run");  }  } |

При применении этого интерфейса класс BaseAction должен будет реализовать как методы и свойства интерфейса IRunAction, так и методы и свойства базового интерфейса IAction, если эти методы и свойства не имеют реализации по умолчанию.

Однако в отличие от классов мы не можем применять к интерфейсам модификатор sealed, чтобы запретить наследование интерфейсов.

Также мы не можем применять к интерфейсам модификатор abstract, поскольку интерфейс фактически, итак, как правило, предоставляет абстрактный функционал, который должен быть реализован в классе или структуре (за исключением методов и свойств с реализацией по умолчанию).

Однако методы интерфейсов могут использовать ключевое слово new для сокрытия методов из базового интерфейса:

|  |
| --- |
| С# |
| class RunAction : IRunAction  {  public void Move()  {  Console.WriteLine("I am running");  }  }    interface IAction  {  void Move();  }  interface IRunAction : IAction  {  new void Move();  } |

Здесь метод Move из IRunAction скрывает метод Move из базового интерфейса IAction. Большого смысла в этом нет, так как в данном случае нечего скрывать, то тем не менее мы так можем делать. А класс RunAction реализует метод Move сразу для обоих интерфейсов.

При наследовании интерфейсов следует учитывать, что, как и при наследовании классов, производный интерфейс должен иметь тот же уровень доступа или более строгий, чем базовый интерфейс. Например:

|  |
| --- |
| С# |
| public interface IAction  {  void Move();  }  internal interface IRunAction : IAction  {  void Run();  } |

Но не наоборот. Например, в следующем случае мы получим ошибку, и программа не скомпилируется, так как производный интерфейс имеет менее строгий уровень доступа, нежели базовый:

|  |
| --- |
| С# |
| internal interface IAction  {      void Move();  }  public interface IRunAction : IAction   // ошибка IRunAction может быть только internal  {      void Run();  } |

## Интерфейсы в обобщениях

### Интерфейсы как ограничения обобщений

Интерфейсы могут выступать в качестве ограничений обобщений. При этом если в качестве ограничения можно указать только один класс, то интерфейсов можно указать несколько.

Допустим, у нас есть следующие интерфейсы и класс, который их реализует:

|  |
| --- |
| С# |
| interface IAccount  {  int CurrentSum { get; } // Текущая сумма на счету  void Put(int sum); // Положить деньги на счет  void Withdraw(int sum); // Взять со счета  }  interface IClient  {  string Name { get; set; }  }  class Client : IAccount, IClient  {  int \_sum; // Переменная для хранения суммы  public Client(string name, int sum)  {  Name = name;  \_sum = sum;  }    public string Name { get; set; }  public int CurrentSum  {  get { return \_sum; }  }  public void Put(int sum)  {  \_sum += sum;  }  public void Withdraw(int sum)  {  if (sum <= \_sum)  {  \_sum -= sum;  }  }  } |

Используем вышеперечисленные интерфейсы в качестве ограничений обобщенного класса:

|  |
| --- |
| С# |
| class Transaction<T> where T: IAccount, IClient  {  public void Operate(T acc1, T acc2, int sum)  {  if(acc1.CurrentSum >= sum)  {  acc1.Withdraw(sum);  acc2.Put(sum);  Console.WriteLine($"{acc1.Name} : {acc1.CurrentSum}\n{acc2.Name} : {acc2.CurrentSum}");  }  }  } |

В данном случае параметр T представляет тип, который который реализует сразу два интерфейса IAccount и IClient. Например, выше определен класс Client, который реализует оба интерфейса, поэтому мы можем данным типом типизировать объекты Transaction:

|  |
| --- |
| С# |
| Client account1 = new Client("Tom", 200);  Client account2 = new Client("Bob", 300);  Transaction<Client> transaction = new Transaction<Client>();  transaction.Operate(account1, account2, 150); |

Также параметр T может представлять интерфейс, который наследуется от обоих интерфейсов:

|  |
| --- |
| С# |
| interface IClientAccount : IAccount, IClient  {    }  class ClientAccount : IClientAccount  {  int \_sum;  public ClientAccount(string name, int sum)  {  \_sum = sum; Name = name;  }  public int CurrentSum { get { return \_sum; } }    public string Name { get; set; }    public void Put(int sum)  {  \_sum += sum;  }  public void Withdraw(int sum)  {  if (\_sum >= sum) \_sum -= sum;  }  } |

В этом случае объекты Transaction мы можем типизировать типом IClientAccount:

|  |
| --- |
| С# |
| IClientAccount account3 = new ClientAccount("Alice", 400);  IClientAccount account4 = new ClientAccount("Kate", 500);  Transaction<IClientAccount> operation = new Transaction<IClientAccount>();  operation.Operate(account3, account4, 200); |

### Обобщенные интерфейсы

Как и классы, интерфейсы могут быть обобщенными:

|  |
| --- |
| С# |
| interface IUser<T>  {  T Id { get; }  }  class User<T> : IUser<T>  {  T \_id;  public User(T id)  {  \_id = id;  }  public T Id { get { return \_id; } }  } |

Интерфейс IUser типизирован параметром T, который при реализации интерфейса используется в классе User. В частности, переменная \_id определена как T, что позволяет нам использовать для id различные типы.

Определим две реализации: одна в качестве параметра будет использовать тип int, а другая - тип string:

|  |
| --- |
| С# |
| IUser<int> user1 = new User<int>(6789);  Console.WriteLine(user1.Id); // 6789    IUser<string> user2 = new User<string>("12345");  Console.WriteLine(user2.Id); // 12345 |

Также при реализации интерфейса мы можем явным образом указать, какой тип будет использоваться для параметра T:

|  |
| --- |
| С# |
| class IntUser : IUser<int>  {      int \_id;      public IntUser(int id)      {          \_id = id;      }      public int Id { get { return \_id; } }  } |

## Копирование объектов. Интерфейс ICloneable

Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности:

|  |
| --- |
| С# |
| class Program  {  static void Main(string[] args)  {  Person p1 = new Person { Name="Tom", Age = 23 };  Person p2 = p1;  p2.Name = "Alice";  Console.WriteLine(p1.Name); // Alice    Console.Read();  }  }    class Person  {  public string Name { get; set; }  public int Age { get; set; }  } |

В данном случае объекты p1 и p2 будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной p2 затронут также и переменную p1.

Чтобы переменная p2 указывала на новый объект, но со значениями из p1, мы можем применить клонирование с помощью реализации интерфейса ICloneable:

|  |
| --- |
| С# |
| public interface ICloneable  {      object Clone();  } |

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

|  |
| --- |
| С# |
| class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public object Clone()      {          return new Person { Name = this.Name, Age = this.Age };      }  } |

Использование:

|  |
| --- |
| С# |
| class Program  {  static void Main(string[] args)  {  Person p1 = new Person { Name="Tom", Age = 23 };  Person p2 = (Person)p1.Clone();  p2.Name = "Alice";  Console.WriteLine(p1.Name); // Tom    Console.Read();  }  } |

Теперь все нормально копируется, изменения в свойствах p2 не сказываются на свойствах в p1.

Для сокращения кода копирования мы можем использовать специальный метод MemberwiseClone(), который возвращает копию объекта:

|  |
| --- |
| С# |
| class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public object Clone()  {  return this.MemberwiseClone();  }  } |

Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:

|  |
| --- |
| С# |
| class Person : ICloneable  {  public string Name { get; set; }  public int Age { get; set; }  public Company Work { get; set; }    public object Clone()  {  return this.MemberwiseClone();  }  }    class Company  {  public string Name { get; set; }  } |

В этом случае при копировании новая копия будет указывать на тот же объект Company:

|  |
| --- |
| С# |
| Person p1 = new Person { Name="Tom", Age = 23, Work= new Company { Name = "Microsoft" } };  Person p2 = (Person)p1.Clone();  p2.Work.Name = "Google";  p2.Name = "Alice";  Console.WriteLine(p1.Name); // Tom  Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft |

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять глубокое копирование:

|  |
| --- |
| С# |
| class Person : ICloneable  {      public string Name { get; set; }      public int Age { get; set; }      public Company Work { get; set; }        public object Clone()      {          Company company = new Company { Name = this.Work.Name };          return new Person          {              Name = this.Name,              Age = this.Age,              Work = company          };      }  }    class Company  {      public string Name { get; set; }  } |

## Сортировка объектов. Интерфейс IComparable

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:

|  |
| --- |
| С# |
| int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };  Array.Sort(numbers);  foreach (int n in numbers)  Console.WriteLine(n); |

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:

|  |
| --- |
| С# |
| public interface IComparable  {      int CompareTo(object o);  } |

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
* Равен нулю. Значит, оба объекта равны
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

|  |
| --- |
| С# |
| class Person : IComparable  {      public string Name { get; set; }      public int Age { get; set; }      public int CompareTo(object o)      {          Person p = o as Person;          if (p != null)              return this.Name.CompareTo(p.Name);          else              throw new Exception("Невозможно сравнить два объекта");      }  } |

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

|  |
| --- |
| С# |
| Person p1 = new Person { Name = "Bill", Age = 34 };  Person p2 = new Person { Name = "Tom", Age = 23 };  Person p3 = new Person { Name = "Alice", Age = 21 };    Person[] people = new Person[] { p1, p2, p3 };  Array.Sort(people);    foreach(Person p in people)  {      Console.WriteLine($"{p.Name} - {p.Age}");  } |

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

|  |
| --- |
| С# |
| class Person : IComparable<Person>  {      public string Name { get; set; }      public int Age { get; set; }      public int CompareTo(Person p)      {          return this.Name.CompareTo(p.Name);      }  } |

### Применение компаратора

Кроме интерфейса IComparable платформа .NET также предоставляет интерфейс IComparer:

|  |
| --- |
| С# |
| interface IComparer  {      int Compare(object o1, object o2);  } |

Метод Compare предназначен для сравнения двух объектов o1 и o2. Он также возвращает три значения, в зависимости от результата сравнения: если первый объект больше второго, то возвращается число больше 0, если меньше - то число меньше нуля; если оба объекта равны, возвращается ноль.

Создадим компаратор объектов Person. Пусть он сравнивает объекты в зависимости от длины строки - значения свойства Name:

|  |
| --- |
| С# |
| class PeopleComparer : IComparer<Person>  {      public int Compare(Person p1, Person p2)      {          if (p1.Name.Length > p2.Name.Length)              return 1;          else if (p1.Name.Length < p2.Name.Length)              return -1;          else              return 0;      }  } |

В данном случае используется обобщенная версия интерфейса IComparer, чтобы не делать излишних преобразований типов. Применение компаратора:

|  |
| --- |
| С# |
| Person p1 = new Person { Name = "Bill", Age = 34 };  Person p2 = new Person { Name = "Tom", Age = 23 };  Person p3 = new Person { Name = "Alice", Age = 21 };    Person[] people = new Person[] { p1, p2, p3 };  Array.Sort(people, new PeopleComparer());    foreach(Person p in people)  {      Console.WriteLine($"{p.Name} - {p.Age}");  } |

Объект компаратора указывается в качестве второго параметра метода Array.Sort(). При этом не важно, реализует ли класс Person интерфейс IComparable или нет. Правила сортировки, установленные компаратором, будут иметь больший приоритет. В начале будут идти объекты Person, у которых имена меньше, а в конце - у которых имена длиннее:

|  |
| --- |
| С# |
| Tom - 23  Bill - 34  Alice - 21 |

## Ковариантность и контравариантность обобщенных интерфейсов

Понятия ковариантности и контравариантности связаны с возможностью использовать в приложении вместо некоторого типа другой тип, который находится ниже или выше в иерархии наследования.

Имеется три возможных варианта поведения:

* **Ковариантность**: позволяет использовать более конкретный тип, чем заданный изначально
* **Контравариантность**: позволяет использовать более универсальный тип, чем заданный изначально
* **Инвариантность**: позволяет использовать только заданный тип

C# позволяет создавать ковариантные и контравариантные обобщенные интерфейсы. Эта функциональность повышает гибкость при использовании обобщенных интерфейсов в программе. По умолчанию все обобщенные интерфейсы, например, IAccout<T> являются инвариантными.

Для рассмотрения ковариантных и контравариантных интерфейсов возьмем следующие классы:

|  |
| --- |
| С# |
| class Account  {      public virtual void DoTransfer(int sum)      {      Console.WriteLine($"Клиент положил на счет {sum} долларов");      }  }  class DepositAccount : Account  {      public override void DoTransfer(int sum)      {          Console.WriteLine($"Клиент положил на депозитный счет {sum} долларов");      }  } |

Здесь определен класс обычного счета - Account и унаследованный от него класс DepositAccount. В классе Account определен метод, который выполняет условную операцию с счетом. Класс DepositAccount немного переопределяет этот интерфейс.

### Ковариантные интерфейсы

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово out. Например:

|  |
| --- |
| С# |
| interface IBank<out T>  {  T CreateAccount(int sum);  }    class Bank<T> : IBank<T> where T : Account, new()  {  public T CreateAccount(int sum)  {  T acc = new T(); // создаем счет  acc.DoTransfer(sum);  return acc;  }  } |

Обобщенный интерфейс IBank определяет метод CreateAccount для создания счета. При этом на момент определения интерфейса мы не знаем, какой тип будет представлять счет. Ключевое слово out в определении интерфейса указывает, что данный интерфейс будет ковариантным.

Класс Bank, который представляет условный банк, реализует этот интерфейс и возвращает из метода CreateAccount объект, который представляет либо класс Account, либо один из его наследников.

Применим данные типы в программе:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  IBank<DepositAccount> depositBank = new Bank<DepositAccount>();  Account acc1 = depositBank.CreateAccount(34);    IBank<Account> ordinaryBank = new Bank<DepositAccount>();  // или так  // IBank<Account> ordinaryBank = depositBank;  Account acc2 = ordinaryBank.CreateAccount(45);    Console.Read();  } |

То есть мы можем присвоить более общему типу IBank<Account> объект более конкретного типа IBank<DepositAccount> или Bank<DepositAccount>.

В то же время если бы мы не использовали ключевое слово out:

|  |
| --- |
| С# |
| interface IBank<out T> |

то мы столкнулись бы с ошибкой в строке

|  |
| --- |
| С# |
| IBank<Account> ordinaryBank = depositBank; |

Поскольку в этом случае невозможно было бы привести объект Bank<DepositAccount> к типу IBank<Account>

При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

### Контравариантные интерфейсы

Для создания контравариантного интерфейса надо использовать ключевое слово in. Например, возьмем те же классы Account и DepositAccount и определим следующие типы:

|  |
| --- |
| С# |
| interface ITransaction<in T>  {  void DoOperation(T account, int sum);  }    class Transaction<T> : ITransaction<T> where T : Account  {  public void DoOperation(T account, int sum)  {  account.DoTransfer(sum);  }  } |

Здесь определен интерфейс ITransaction, который представляет условную банковскую операцию. Ключевое слово in в определении интерфейса указывает, что этот интерфейс - контравариантный. Интерфейс определяет метод DoOperation, который принимает некоторый счет и выполняет с ним операцию.

Класс Transaction реализует этот интерфейс и реализует его метод DoOperation.

Применим эти типы в программе:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  ITransaction<Account> accTransaction = new Transaction<Account>();  accTransaction.DoOperation(new Account(), 400);    ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>();  depAccTransaction.DoOperation(new DepositAccount(), 450);    Console.Read();  } |

Так как интерфейс ITransaction использует универсальный параметр с ключевым словом in, то он является контравариантным, поэтому в коде мы можем объект Transaction<Account> привести к типу ITransaction<DepositAccount>:

|  |
| --- |
| С# |
| ITransaction<DepositAccount> depAccTransaction = new Transaction<Account>(); |

Если бы ключевое слово in не использовалось бы, то мы не смогли бы выполнить подобное приведение. То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к аргументам, используемым в качестве возвращаемых типов.

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Решите задание используя интерфейсы
2. Спроектируйте блок схему к программе
3. Отобразить в отчете проделанные шаги
4. Ответить на контрольные вопросы
5. Написать вывод о проделанной работе

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

|  |
| --- |
| Вариант 1 – Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о клиентах автомойки (ФИО, дата записи на мойку, день недели, модель машины). |
| Вариант 2 – Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о заказах в KFC (Дата заказа, номер транзакции, список заказа, общая стоимость). |
| Вариант 3 – Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о студентах техникума (Номер группы, ФИО, курс, номер зачетки, дата поступление, дата окончания обучения). |
| Вариант 4 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о семенах растений на складе (Инвентаризационный номер, наименование, тип растений, дата поступления на склад, сезон посадки, срок созревания, климат). |
| Вариант 5 – Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о кабинете (Инвентаризационный номер, наименование, тип мебели, количество, срок проверки инвентаризации). |
| Вариант 6 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию об абонентах провайдера (Номер договора, дата подключения, ФИО клиента, тип подключения, IP-адрес, скорость подключения). |
| Вариант 7 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию об авиабилетов (Номер билета, номер рейса, время и дата отправления, место отправки, место прибытия). |
| Вариант 8 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию об учебных заведений (Номер учебного заведения, номер аккредитации, наименования, срок действия лицензии, регион, тип образования). |
| Вариант 9 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о товарах в магазине (Номер артикля, наименования, дата поступления, срок годности, тип продукта, изготовитель). |
| Вариант 10 - Реализуйте программу используя интерфейсы. Создайте программу, которая содержит информацию о комплектующих ПК (Номер сборки, модель процессора, оперативной памяти, постоянной памяти, блок-питания и т.д). |

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Где применяются интерфейсы?
2. Как работает наследование интерфейсов?
3. Как работает компаратор?

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Официальная документация Microsoft C# — Текст: электронный // Microsoft [сайт]. — URL: https://docs.microsoft.com/ (дата обращения: 12.03.2020).
2. Сообщество IT-специалистов — Текст: электронный // Habr [сайт]. — URL: https://habr.com/ (дата обращения: 12.03.2020).
3. Сайт о программирование / — Текст: электронный // Metanit [сайт]. — https://metanit.com/ (дата обращения: 12.03.2020).
4. С# Tutotial — Текст: электронный // Tutotial [сайт]. — URL: https://csharp.net-tutorials.com/ (дата обращения: 13.03.2020).

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Задачи
   2. Код
   3. Блок-схемы
   4. Результат работы
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по учебной-практической работе №5

учебной практике   
*УП 05. Программирование*

*Тема: «Интерфейсы»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель

Владимирова А.В.

г. Томск – 2020 г