**ЛАБОРАТОРНА РОБОТА №5**

**Тема: УСПАДКУВАННЯ. АБСТАКТНІ КЛАСИ. ІНТЕРФЕЙСИ**

**Мета**: придбати практичні навички роботи створення базоаового та похідних класів, перевизначення методів у класі, методів перевантаження унарних, бінарних, логічних операторів, операторів порівняння.

**Постановка задачі:**

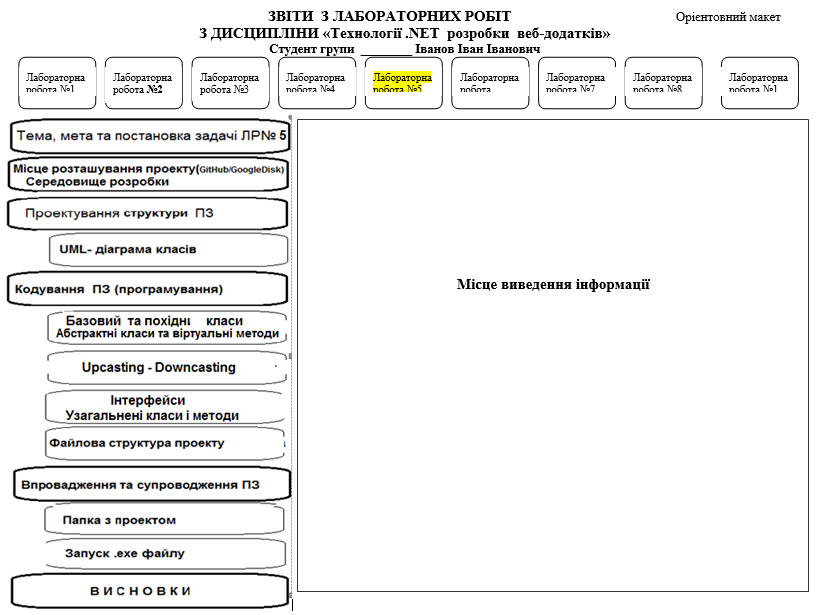
1. Доповнити програмну реалізацію класів методами для перевантаження унарних, бінарних, логічних операторів, операторів порівняння.
2. Протестувати програму. Зробити висновки.

**ХІД ВИКОНАННЯ РОБОТИ**

1. У пункті меню **«*Тема, мета та постановка задачі ЛР №4* »** звітного HTML-документу розмістити :
   1. Тему лабораторної роботи №5
   2. Мету лабораторної роботи №5
   3. Постановку задачі лабораторної роботи №5
   4. Список базових, похідних, абстрактних класів, інтерфейсів, віртуальних та абстрактних методів, які реалізуються у даній лабораторній роботі. Написати призначення ціх класів та методів.
2. У пункті меню «*Місце розташування проекту.Середовище розробки*» вказатим **місце розташування проекту** (шлях до папки з проектом). Це може бути GitHub або GoogeDisk. Також написати **середовище розробки програми.**
3. У пункі «Проектування структури ПЗ» розмістити діаграму класів, попередньо добавивши базові, похідні класи, інтерфейси
4. У пункті меню « *Кодування ПЗ (програмування)» розмістити програмні коди та результати їх виконання :*
   1. базового та похідних класів;
   2. абстрактних класів, віртуальних та абстрактних методів;
   3. висхідних (Upcasting) та спадних (Downcasting) перетворень (у власному проекті або на окремій WEB-сторінці);
   4. інтерфейсів;
   5. узагальнені класи та методи;
5. У пункті меню «*Впровадження та супроводження ПЗ*» розмістити посилання на папку з проектом, усіма файлами проекту, виконувальним файлом.ехе, з можливістю відкриття проекту зі звіту.
6. У пункті меню «*ВИСНОВКИ* »написати висновки по лабораторній роботи
7. Самостійно відпрацювати матеріал, приведенний у посібникау Герберта Шилдта «С# 4.0» Полное руководство.

Відпрацювати матеріал за посиланням <https://metanit.com/sharp/tutorial/>

1. Для оформлення звітного HTML-документу приведен орієнтовний макет.



**ПРИМІТКА**:

Звіт з лабораторних робіт слід підготувати у вигляді **гіпертекстового документа у форматі html.** Документ має містити меню, яке включає команди, що подані нижче. Слід реалізувати запуск програм на виконання з гіпертекстового документа.

Звіт та проекти лабораторних робіт слід записати на Гугл-Диск а. На диску має бути файл readme.tхt, який містить відомості про автора звіту та проектів.

**Титульна html- сторінка.**

Назва роботи

Автор (ПІБ, група, курс, № заліковки)

Фото

Рік навчання

**ТЕОРЕТИЧНІ ВІДОМОСТІ**

## ***Наслідуванння***

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

Пусть у нас есть следующий класс Person, который описывает отдельного человека:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Person  {      private string \_name;        public string Name      {          get { return \_name; }          set { \_name = value; }      }      public void Display()      {          Console.WriteLine(Name);      }  } |

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс Employee. Поскольку этот класс будет реализовывать тот же функционал, что и класс Person, так как сотрудник - это также и человек, то было бы рационально сделать класс Employee производным (или наследником, или подклассом) от класса Person, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

|  |  |
| --- | --- |
| 1  2  3  4 | class Employee : Person  {    } |

После двоеточия мы указываем базовый класс для данного класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же свойства, методы, поля, которые есть в классе Person. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение **is-a** (является), объект класса Employee также является объектом класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      Person p = new Person { Name = "Tom"};      p.Display();      p = new Employee { Name = "Sam" };      p.Display();      Console.Read();  } |

И поскольку объект Employee является также и объектом Person, то мы можем так определить переменную: Person p = new Employee().

По умолчанию все классы наследуются от базового класса **Object**, даже если мы явным образом не устанавливаем наследование. Поэтому выше определенные классы Person и Employee кроме своих собственных методов, также будут иметь и методы класса Object: ToString(), Equals(), GetHashCode() и GetType().

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

* Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
* При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. То есть, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

Однако следует также учитывать, что если базовый и производный класс находятся в разных сборках (проектах), то в этом случае производый класс может наследовать только от класса, который имеет модификатор public.

* Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

|  |  |
| --- | --- |
| 1  2  3 | sealed class Admin  {  } |

* Нельзя унаследовать класс от статического класса.

### **Доступ к членам базового класса из класса-наследника**

Вернемся к нашим классам Person и Employee. Хотя Employee наследует весь функционал от класса Person, посмотрим, что будет в следующем случае:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Employee : Person  {      public void Display()      {          Console.WriteLine(\_name);      }  } |

Этот код не сработает и выдаст ошибку, так как переменная \_name объявлена с модификатором private и поэтому к ней доступ имеет только класс Person. Но зато в классе Person определено общедоступное свойство Name, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Employee : Person  {      public void Display()      {          Console.WriteLine(Name);      }  } |

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **private protected** (если базовый и производный класс находятся в одной сборке), **public**, **internal** (если базовый и производный класс находятся в одной сборке), **protected** и **protected internal**.

### **Ключевое слово base**

Теперь добавим в наши классы конструкторы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class Person  {      public string Name { get;  set; }        public Person(string name)      {          Name = name;      }        public void Display()      {          Console.WriteLine(Name);      }  }    class Employee : Person  {      public string Company { get; set; }        public Employee(string name, string company)          : base(name)      {          Company = company;      }  } |

Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова **base** мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      Person p = new Person("Bill");      p.Display();      Employee emp = new Employee ("Tom", "Microsoft");      emp.Display();      Console.Read();  } |

### **Конструкторы в производных классах**

Конструкторы не передаются производному классу при наследовании. И если в базовом классе **не определен** конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом Person), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово base. Например, из класса Employee уберем определение конструктора:

|  |  |
| --- | --- |
| 1  2  3  4 | class Employee : Person  {      public string Company { get; set; }  } |

В данном случае мы получим ошибку, так как класс Employee не соответствует классу Person, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string name, string company)  {      Name = name;      Company = company;  } |

То есть в классе Employee через ключевое слово **base** надо явным образом вызвать конструктор класса Person:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string name, string company)          : base(name)  {      Company = company;  } |

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class Person  {      // остальной код класса      // конструктор по умолчанию      public Person()      {          FirstName = "Tom";          Console.WriteLine("Вызов конструктора без параметров");      }  } |

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

|  |  |
| --- | --- |
| 1  2  3  4 | public Employee(string company)  {      Company = company;  } |

Фактически был бы эквивалентен следующему конструктору:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public Employee(string company)      :base()  {      Company = company;  } |

### **Порядок вызова конструкторов**

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных. Например, возьмем следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Person  {      string name;      int age;        public Person(string name)      {          this.name = name;          Console.WriteLine("Person(string name)");      }      public Person(string name, int age) : this(name)      {          this.age = age;          Console.WriteLine("Person(string name, int age)");      }  }  class Employee : Person  {      string company;        public Employee(string name, int age, string company) : base(name, age)      {          this.company = company;          Console.WriteLine("Employee(string name, int age, string company)");      }  } |

При создании объекта Employee:

|  |  |
| --- | --- |
| 1 | Employee tom = new Employee("Tom", 22, "Microsoft"); |

Мы получим следующий консольный вывод:

Person(string name)

Person(string name, int age)

Employee(string name, int age, string company)

В итоге мы получаем следующую цепь выполнений.

1. Вначале вызывается конструктор Employee(string name, int age, string company). Он делегирует выполнение конструктору Person(string name, int age)
2. Вызывается конструктор Person(string name, int age), который сам пока не выполняется и передает выполнение конструктору Person(string name)
3. Вызывается конструктор Person(string name), который передает выполнение конструктору класса System.Object, так как это базовый по умолчанию класс для Person.
4. Выполняется конструктор System.Object.Object(), затем выполнение возвращается конструктору Person(string name)
5. Выполняется тело конструктора Person(string name), затем выполнение возвращается конструктору Person(string name, int age)
6. Выполняется тело конструктора Person(string name, int age), затем выполнение возвращается конструктору Employee(string name, int age, string company)
7. Выполняется тело конструктора Employee(string name, int age, string company). В итоге создается объект Employee

## ***перетворення типів***

Припустимо, у нас є наступна ієрархія класів:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class Person  {      public string Name { get; set; }      public Person(string name)      {          Name = name;      }      public void Display()      {          Console.WriteLine($"Person {Name}");      }  }    class Employee : Person  {      public string Company { get; set; }      public Employee(string name, string company) : base(name)      {          Company = company;      }  }    class Client : Person  {      public string Bank { get; set; }      public Client(string name, string bank) : base(name)      {          Bank = bank;      }  } |

У цій ієрархії класів ми можемо простежити наступну ланцюг успадкування: Object (всі класи неявно успадковуються від типу Object) -> Person -> Employee | Client.

Причому в цій ієрархії класів базові типи знаходяться вгорі, а похідні типи - внизу.

### **Висхідні перетворення. Upcasting**

Об'єкти похідного типу (який знаходиться внизу ієрархії) в той же час представляють і базовий тип. Наприклад, об'єкт Employee в той же час є і об'єктом класу Person. Що в принципі природно, так як кожен співробітник (Employee) є людиною (Person). І ми можемо написати, наприклад, наступним чином:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      Employee employee = new Employee("Tom", "Microsoft");      Person person = employee;   // преобразование от Employee к Person        Console.WriteLine(person.Name);      Console.ReadKey();  } |

В даному випадку змінної person, яка представляє тип Person, присвоюється посилання на об'єкт Employee. Але щоб зберегти посилання на об'єкт одного класу в змінну іншого класу, необхідно виконати перетворення типів - в даному випадку від типу Employee до типу Person. І так як Employee успадковується від класу Person, то автоматично виконується неявне висхідний перетворення - перетворення до типу, які знаходяться вгорі ієрархії класів, тобто до базового класу.

В результаті змінні employee і person вказуватимуть на один і той же об'єкт в пам'яті, але змінної person буде доступна тільки та частина, яка представляє функціонал типу Person.

Подібним чином поізводіт і інші висхідні перетворення:

|  |  |
| --- | --- |
| 1 | Person person2 = new Client("Bob", "ContosoBank");   // преобразование от Client к Person |

Тут змінна person2, яка представляє тип Person, зберігає посилання на об'єкт Client, тому також виконується висхідний неявне перетворення від похідного класу Client до базового типу Person.

Висхідний неявне перетворення відбуватиметься і в наступному випадку:

|  |  |
| --- | --- |
| 1  2  3 | object person1 = new Employee("Tom", "Microsoft");  // от Employee к object  object person2 = new Client("Bob", "ContosoBank");  // от Client к object  object person3 = new Person("Sam");                 // от Person к object |

Так як тип object - базовий для всіх інших типів, то перетворення до нього буде проводитися автоматично.

### **Спадні перетворення. Downcasting**

Але крім висхідних перетворень від похідного до базового типу є хто сходить перетворення або downcasting - від базового типу до похідного. Наприклад, в наступному коді змінна person зберігає посилання на об'єкт Employee:

|  |  |
| --- | --- |
| 1  2 | Employee employee = new Employee("Tom", "Microsoft");  Person person = employee;   // преобразование от Employee к Person |

І може виникнути питання, чи можна звернутися до функціонала типу Employee через змінну типу Person. Але автоматично такі перетворення не проходять, адже не кожна людина (об'єкт Person) є співробітником підприємства (об'єктом Employee). І для низхідного перетворення необхідно застосувати явне перетворення, вказавши в дужках тип, до якого потрібно виконати перетворення:

|  |  |
| --- | --- |
| 1  2  3  4  5 | Employee employee = new Employee("Tom", "Microsoft");  Person person = employee;   // преобразование от Employee к Person    //Employee employee2 = person;    // так нельзя, нужно явное преобразование  Employee employee2 = (Employee)person;  // преобразование от Person к Employee |

Розглянемо деякі приклади перетворень:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // чтобы обратиться к возможностям типа Employee, приводим объект к типу Employee  Employee emp = (Employee) obj;    // объект Client также представляет тип Person  Person person = new Client("Sam", "ContosoBank");  // преобразование от типа Person к Client  Client client = (Client)person; |

У першому випадку змінної obj присвоєна посилання на об'єкт Employee, тому ми можемо перетворити об'єкт obj до будь-якого типу який розташовується в ієрархії класів між типом object і Employee.

Якщо нам треба звернутися до якимось окремим властивостям або методам об'єкта, то нам необов'язково привласнювати перетворений об'єкт змінної:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // преобразование к типу Person для вызова метода Display  ((Person)obj).Display();  // либо так  // ((Employee)obj).Display();    // преобразование к типу Employee, чтобы получить свойство Company  string comp = ((Employee)obj).Company; |

У той же час необхідно дотримуватися обережності при подібних перетвореннях. Наприклад, що буде в наступному випадку:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // Объект Employee также представляет тип object  object obj = new Employee("Bill", "Microsoft");    // преобразование к типу Client, чтобы получить свойство Bank  string bank = ((Client)obj).Bank; |

В даному випадку ми отримаємо помилку, так як змінна obj зберігає посилання на об'єкт Employee. Даний об'єкт є також об'єктом типів object і Person, тому ми можемо перетворити його до цих типів. Але до типу Client ми перетворити не можемо.

Інший приклад:

|  |  |
| --- | --- |
| 1  2  3  4 | Employee emp = new Person("Tom");   // ! Ошибка    Person person = new Person("Bob");  Employee emp2 = (Employee) person;  // ! Ошибка |

В даному випадку ми намагаємося перетворити об'єкт типу Person до типу Employee, а об'єкт Person не є об'єктом Employee.

Існує ряд способів, щоб уникнути подібних помилок перетворення.

### **способи перетворень**

По-перше, можна використовувати ключове слово **as** . За допомогою нього програма намагається перетворити вираз до певного типу, при цьому не викидає виключення. У разі невдалого перетворення вираз буде містити значення null:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person person = new Person("Tom");  Employee emp = person as Employee;  if (emp == null)  {      Console.WriteLine("Преобразование прошло неудачно");  }  else  {      Console.WriteLine(emp.Company);  } |

Другий спосіб полягає в відловлюванні виключення **InvalidCastException** , яке виникне в результаті перетворення:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person person = new Person("Tom");  try  {      Employee emp = (Employee)person;      Console.WriteLine(emp.Company);  }  catch (InvalidCastException ex)  {      Console.WriteLine(ex.Message);  } |

Третій спосіб полягає в перевірці допустимості перетворення за допомогою ключового слова **is** :

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Person person = new Person("Tom");  if(person is Employee)  {      Employee emp = (Employee)person;      Console.WriteLine(emp.Company);  }  else  {      Console.WriteLine("Преобразование не допустимо");  } |

Вираз person is Employeeперевіряє, чи є змінна person об'єктом типу Employee. Але так як в даному випадку явно не є, то така перевірка поверне значення false, і перетворення не спрацює.