¶Московский авиационный институт

(Национальный исследовательский университет)

**Факультет информационных технологий и прикладной математики**

Кафедра вычислительной математики и программирования

**Лабораторные работы 1 — 9 по курсу ООП:  
основы программирования на С#**

0.Перегрузка

1.Агрегация по ссылке

2.Агрегация по значению и вложением

3.Принцип подстановки

к4.Наследование: расширение, спецификация, специализация, конструирование и комбинирование

5.Наследование: комбинирование через общих предков

6.Ассоциация(один к одному, один ко многим)

7.Использование: клиент-сервер, клиент-утилита.

8.Конкретизация: параметров функции, метода, конструктора и атрибутов.

Множественная конкретизация. Конкретизация с ограничениями (оператор Is)

9. АНОНИМНЫЕ ФУНКЦИИ: сигнатура функции (delegate), ЛЯМБДА ВЫРАЖЕНИЕ(=>). Event

Работу выполнил:

Студент группы М8О-205Б-19 Васильев А. В./\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Вариант 4

Руководитель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/Семенов А.С.

Дата:

**0.Перегрузка конструкторов, функций, операторов и операций.**

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_0

{

class Reestr //Перегрузка: конструктор, функции, операции,операторы.

{

public override string ToString()

{

return $"Фамилия: {surn}; Номер в списке: {numb.ToString()}; Год публикации: {publyear.ToString()}; Возраст: {age.ToString()}";

}

string surn = "---" ;

int numb = 0 ;

int publyear = 2020 ;

int age= 0 ;

public Reestr() //конструктор

{

Console.WriteLine("Constructor without parameters");

}

public Reestr(string surn, int numb, int publyear, int age)

{

this.surn = surn;

this.numb = numb;

this.publyear = publyear;

this.age = age;

Console.WriteLine("Constructor with parameters");

}

public void SurAge() //операция

{

Console.WriteLine("Operation without parameters");

Console.Write("{0}\t", surn);

Console.Write("{0}\t", age);

Console.Write("- Значение имени и возраста по умолчанию");

Console.WriteLine();

}

public void SurAge(int NoInd)

{

Console.WriteLine("Operation with parameter");

Console.Write("{0}\t", numb);

Console.WriteLine("{0}\t","- Номер в списке");

Console.WriteLine("");

}

public int Agediff() //функция

{

Console.WriteLine("Function without parameters");

return 2020 - this.publyear;

}

public int Agediff(int NoInd)

{

Console.WriteLine("Function with parameter");

return 2020 - this.publyear + NoInd;

}

public static int operator +(Reestr unknowperson, Reestr homskiy)//оператор

{

Console.WriteLine("Overload:");

return unknowperson.publyear + homskiy.publyear;

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Overloading constructor");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

Reestr unknowperson = new Reestr();

Console.WriteLine(unknowperson.ToString());

Reestr homskiy = new Reestr("Хомский", 1, 1950, 55);

Console.WriteLine(homskiy.ToString());

Console.WriteLine();

Console.WriteLine("Overloading operation");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

unknowperson.SurAge();

homskiy.SurAge(3);

Console.WriteLine("Overloading function");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

Console.WriteLine(unknowperson.Agediff());

Console.WriteLine(homskiy.Agediff(30));

Console.WriteLine();

Console.WriteLine("Overloading operator");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n");

Console.WriteLine( unknowperson + homskiy) ;

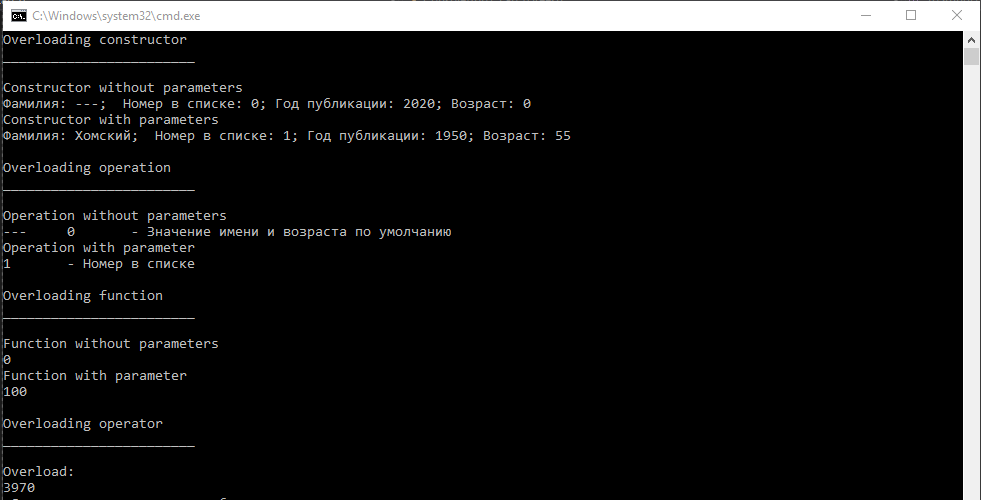
Console.ReadKey();

}

}

}

**Результат работы:**

****

**Вывод:** перегрузка конструкторов, операций, функций и операторов позволяет определить структурный полиморфизм, когда с данным именем существуют несколько функций с разным поведением, а также с различным списком аргументов, который определит, какая функция или операция будет вызываться.

**Лабораторная работа №1. Агрегация по ссылке.**

**А**

**B**

**C**

**J**

**K**

**E**

**F**

public

public

public

public

public

public

public

private

private

private

private

private

private

private

***Рис.1 Агрегация по ссылке.***

**A**

**B**

**C**

**J**

**E**

**K**

**F**

**B(**

**C(**

**A(**

***Рис.2 Память: агрегация по ссылке.***

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_1

{

class A

{

public A(B b, C c, J j)

{

this.b = b;

this.c = c;

this.j = j;

c.cq = 6;

}

~A() { }

public void MethA() { Console.WriteLine("Method A"); }

public B bA

{

set { Console.WriteLine("set b"); b = value; }

get { Console.Write("get b ->"); return b; }

}

public C cA

{

set { Console.WriteLine("set c"); c = value; }

get { Console.Write("get c ->"); return c; }

}

public J jA

{

set { Console.WriteLine("set j"); j = value; }

get { Console.Write("get j ->"); return j; }

}

B b = null;

C c = null;

J j = null;

}

class B

{

public B(K k)

{

this.k = k;

}

~B() { }

public void MethB() { Console.WriteLine("Method B"); }

public K kA

{

set { Console.WriteLine("set k"); k = value; }

get { Console.Write("get k ->"); return k; }

}

K k = null;

}

class C

{

public C(E e, F f)

{

this.e = e;

this.f = f;

}

~C() { }

public void MethC() { Console.WriteLine("Method C"); }

public E eA

{

set { Console.WriteLine("set e"); e = value; }

get { Console.Write("get e ->"); return e; }

}

public F fA

{

set { Console.WriteLine("set f"); f = value; }

get { Console.Write("get f ->"); return f; }

}

public int cq { get; set; }

E e = null;

F f = null;

}

class K

{

public K() { }

~K() { }

public void MethK() { Console.WriteLine("Method K"); }

}

class E

{

public E() { }

~E() { }

public void MethE() { Console.WriteLine("Method E"); }

}

class F

{

public F() { }

~F() { }

public void MethF() { Console.WriteLine("Method F"); }

}

class J

{

public J() { }

~J() { }

public void MethJ() { Console.WriteLine("Method J"); }

}

class Program

{

static void Main(string[] args)

{// Объекты, из которых будут созданы другие.

K k = new K();

E e = new E();

F f = new F();

J j = new J();

// передача вышестоящих объектов в виде параметров в конструкторы.

B b = new B(k);

C c = new C(e, f);

A a = new A(b, c, j);

// вывод атрибута доступа.

Console.WriteLine("Атрибут доступа = {0} ",c.cq);

// передача по ссылке: проверка

Console.WriteLine("Передача по ссылке:");

a.MethA();

a.bA.MethB();

a.cA.MethC();

a.jA.MethJ();

a.bA.kA.MethK();

a.cA.eA.MethE();

a.cA.fA.MethF();

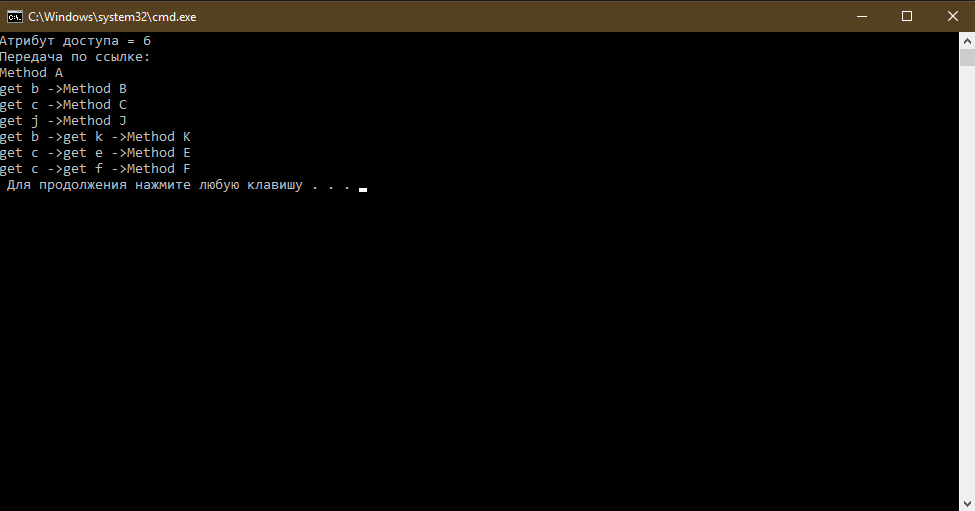
Console.ReadKey();

}

}

}

**Результат работы:**

****

**Вывод:** Объекты всех классов существуют независимо друг от друга. Связывание объектов происходит с помощью конструктора. Например, *b, c, j —* параметры для конструктора класса A; *k* — для конструктора класса B; *e, f —* для конструктора класса C. Объекты могут быть уничтожены по отдельности. Это не нарушит целостность структуры. Если удалить объект *a,* объекты *b, c, j* и т.д будут продолжать существовать и дальше. При агрегации по ссылке каждый этап сборки целого объекта описывается вручную, в отличие от агрегации по значению и вложению.

**Лабораторная работа №2. Агрегация по значению и вложению.**

**Агрегация по значению.**

**А**

**B**

**C**

**J**

**K**

**E**

public

public

public

public

private

private

private

private

**F**

public

public

private

private

public

private

**b**

**c**

**j**

**k**

**e**

**f**

***Рис1. Агрегация по значению.***

***Рис.2 Память: агрегация по значению.***

Ось объектов

t

**a**

**b**

**k**

**c**

**j**

**e**

**f**

**A(**

**B(**

**K(**

**C(**

**E(**

**F(**

**J(**

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab2\_Agr\_po\_Znach

{

class A

{

public A() { c.cq = 6; }

~A() { }

public void MethA() { Console.WriteLine("Method A"); }

public B bA

{

set { Console.WriteLine("set b"); b = value; }

get { Console.Write("get b ->"); return b; }

}

public C cA

{

set { Console.WriteLine("set c"); c = value; }

get { Console.Write("get c ->"); return c; }

}

public J jA

{

set { Console.WriteLine("set j"); j = value; }

get { Console.Write("get j ->"); return j; }

}

B b = new B();

C c = new C();

J j = new J();

}

class B

{

public B() { }

~B() { }

public void MethB() { Console.WriteLine("Method B"); }

public K kA

{

set { Console.WriteLine("set k"); k = value; }

get { Console.Write("get k ->"); return k; }

}

K k = new K();

}

class C

{

public C() { this.cq = 6; }

~C() { }

public void MethC() { Console.WriteLine("Method C"); }

public E eA

{

set { Console.WriteLine("set e"); e = value; }

get { Console.Write("get e ->"); return e; }

}

public F fA

{

set { Console.WriteLine("set f"); f = value; }

get { Console.Write("get f ->"); return f; }

}

public int cq { get; set; }

E e = new E();

F f = new F();

}

class K

{

public K() { }

~K() { }

public void MethK() { Console.WriteLine("Method K"); }

}

class E

{

public E() { }

~E() { }

public void MethE() { Console.WriteLine("Method E"); }

}

class F

{

public F() { }

~F() { }

public void MethF() { Console.WriteLine("Method F"); }

}

class J

{

public J() { }

~J() { }

public void MethJ() { Console.WriteLine("Method J"); }

}

class Program

{

static void Main(string[] args)

{

//C c = new C();

A a = new A();

a.MethA();

a.bA.MethB();

a.cA.MethC();

a.jA.MethJ();

a.bA.kA.MethK();

a.cA.eA.MethE();

a.cA.fA.MethF();

//Console.WriteLine(a.cA.cq);

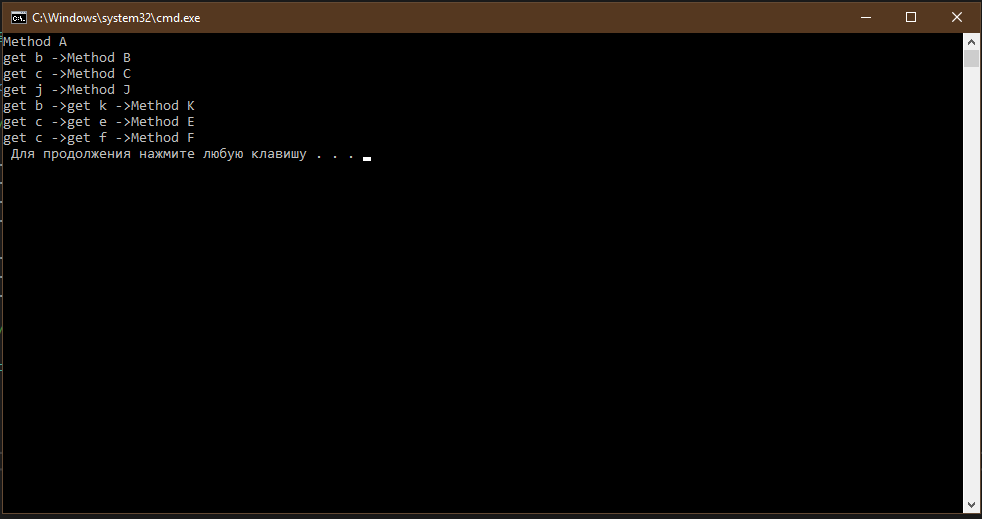
Console.ReadKey();

}

}

}

**Результат работы:**

****

**Вывод:** при агрегации по значению все объекты класса существуют внутри объявленного класса. При таком виде агрегации невозможно удалить объекты, являющиеся частью объекта первого по иерархии класса. Например, *b, c, j —* части объекта *а* класса А(первый класс по иерархии); эти части создаются только при вызове конструктора класса A, а уничтожаются — при вызове деструктора А. При агрегации по значению целый объект создается автоматически без необходимости описания каждого этапа сборки.

**Агрегация по вложению.**

**A**

**B**

**C**

**J**

**K**

**E**

**F**

**b**

**c**

**j**

**k**

**e**

**f**

public

public

public

public

private

private

private

private

private

public

public

public

private

private

***Рис.1 Диаграмма классов: агрегация по вложению.***

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab2\_Agr\_po\_Vlozh

{

class A

{

public A() { c.cq = 5; }

~A() { }

public void MethA() { Console.WriteLine("Method A"); }

public B bA

{

set { Console.WriteLine("set b"); b = value; }

get { Console.Write("get b ->"); return b; }

}

public C cA

{

set { Console.WriteLine("set c"); c = value; }

get { Console.Write("get c ->"); return c; }

}

public J jA

{

set { Console.WriteLine("set j"); j = value; }

get { Console.Write("get j ->"); return j; }

}

B b = new B();

C c = new C();

J j = new J();

public class B // вложенный класс B в A

{

public B() { }

~B() { }

public void MethB() { Console.WriteLine("Method B"); }

public K kA

{

set { Console.WriteLine("set k"); k = value; }

get { Console.Write("get k ->"); return k; }

}

K k = new K();

public class K // вложенный класс K в B

{

public K() { }

~K() { }

public void MethK() { Console.WriteLine("Method K"); }

}

}

public class C // вложенный класс C в A

{

public C() { this.cq = 6; }

~C() { }

public void MethC() { Console.WriteLine("Method C"); }

public E eA

{

set { Console.WriteLine("set e"); e = value; }

get { Console.Write("get e ->"); return e; }

}

public F fA

{

set { Console.WriteLine("set f"); f = value; }

get { Console.Write("get f ->"); return f; }

}

public int cq { get; set; }

E e = new E();

F f = new F();

public class E // вложенный класс E в C

{

public E() { }

~E() { }

public void MethE() { Console.WriteLine("Method E"); }

}

public class F // вложенный класс F в C

{

public F() { }

~F() { }

public void MethF() { Console.WriteLine("Method F"); }

}

}

public class J // вложенный класс J в A

{

public J() { }

~J() { }

public void MethJ() { Console.WriteLine("Method J"); }

}

}

class Program

{

static void Main(string[] args)

{

A a = new A();

a.MethA();

a.bA.MethB();

a.cA.MethC();

a.jA.MethJ();

a.bA.kA.MethK();

a.cA.eA.MethE();

a.cA.fA.MethF();

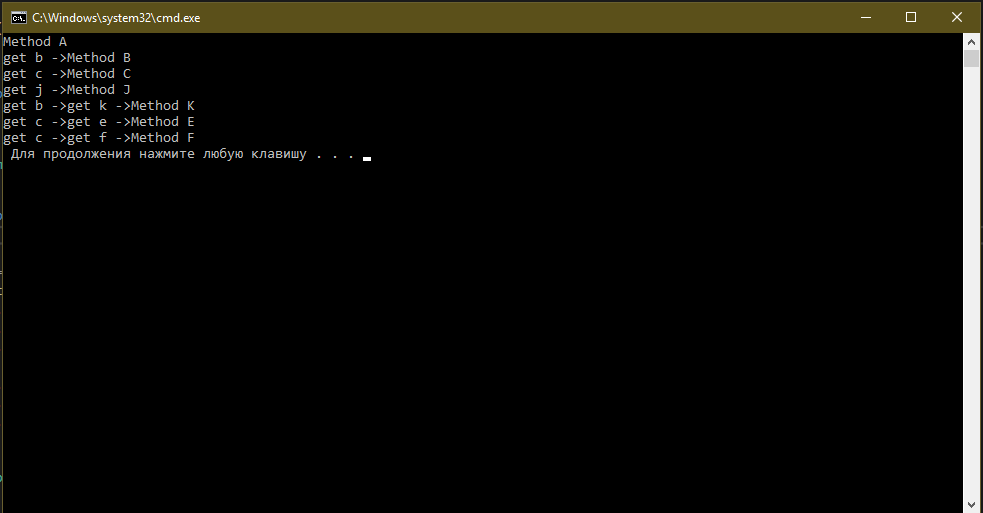
Console.ReadKey();

}

}

}

**Результат работы:**

****

**Вывод:** при агрегации вложением определение классов происходит внутри классов, стоящих выше по иерархии. Все объекты создаваемого класса существуют внутри него самого. Как и в случае агрегации по значению уничтожение объектов, невозможно без уничтожения класса, стоящего выше по иерархии. При агрегации по вложению целый объект создается автоматически без необходимости описания каждого этапа сборки.

**Лабораторная работа №3. Принцип подстановки.**

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab2\_Agr\_po\_Vlozh

{

class A

{

public A() { c.cq = 5; }

~A() { }

public void MethA() { Console.WriteLine("Method A"); }

public B bA

{

set { Console.WriteLine("set b"); b = value; }

get { Console.Write("get b ->"); return b; }

}

public C cA

{

set { Console.WriteLine("set c"); c = value; }

get { Console.Write("get c ->"); return c; }

}

public J jA

{

set { Console.WriteLine("set j"); j = value; }

get { Console.Write("get j ->"); return j; }

}

B b = new B();

C c = new C();

J j = new J();

public class B // вложенный класс B в A

{

public B() { }

~B() { }

public void MethB() { Console.WriteLine("Method B"); }

public K kA

{

set { Console.WriteLine("set k"); k = value; }

get { Console.Write("get k ->"); return k; }

}

K k = new K();

public class K // вложенный класс K в B

{

public K() { }

~K() { }

public void MethK() { Console.WriteLine("Method K"); }

}

}

public class C // вложенный класс C в A

{

public C() { this.cq = 6; }

~C() { }

public void MethC() { Console.WriteLine("Method C"); }

public E eA

{

set { Console.WriteLine("set e"); e = value; }

get { Console.Write("get e ->"); return e; }

}

public F fA

{

set { Console.WriteLine("set f"); f = value; }

get { Console.Write("get f ->"); return f; }

}

public int cq { get; set; }

E e = new E();

F f = new F();

public class E // вложенный класс E в C

{

public E() { }

~E() { }

public void MethE() { Console.WriteLine("Method E"); }

}

public class F // вложенный класс F в C

{

public F() { }

~F() { }

public void MethF() { Console.WriteLine("Method F"); }

}

}

public class J // вложенный класс J в A

{

public J() { }

~J() { }

public void MethJ() { Console.WriteLine("Method J"); }

}

}

class Program

{

static void Main(string[] args)

{

A a = new A();

a.MethA();

a.bA.MethB();

a.cA.MethC();

a.jA.MethJ();

a.bA.kA.MethK();

a.cA.eA.MethE();

a.cA.fA.MethF();

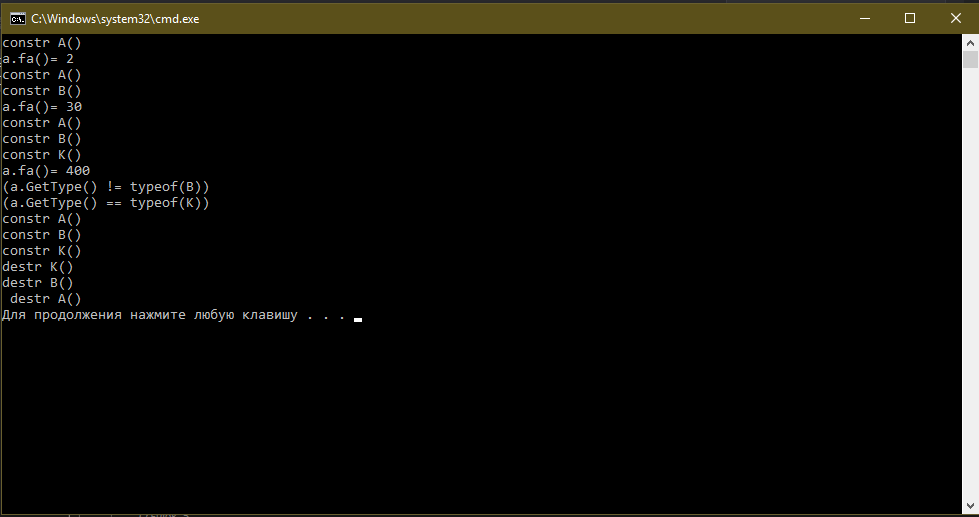
Console.ReadKey();

}

}

}

**Результат работы:**

****

**Вывод :** в этой программе используется метод подстановки и метод замещения. **Принцип подстановки:** вместо объекта суперкласса можно подставить объект подкласса. **Принцип замещения:** функцию суперкласса можно заменить функцией подкласса. Ключевое слово virtual используется для изменения объявлений методов, свойств, в производном классе. Например, этот метод может быть переопределен любым наследующим его классом: модификатор override требуется изменения реализации унаследованного метода, свойства.

**Лабораторная работа №4. Наследование: расширение, спецификация, специализация, конструирование, комбинирование.**

**Наследование: расширение, спецификация, специализация, конструирование**

**А**

**B**

**C**

**J**

**K**

**E**

**F**

public

public

public

public

public

public

public

private

private

private

private

private

private

private

public

private

**D**

**V**

public

private

**Exp**

**Sa**

**Sa**

**Cn**

**Sp**

**Si**

**Ex – расширение B -> A**

**Sa- Спецификация: Abstract class C**

**Si-Спецификация : Interface J**

**Sp-Специализация K->B**

**Cn – конструирование V->K**

*Рис. 1 Наследование: расширение, спецификация, специализация, конструирование.*

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading;

namespace ConsoleApp1\_Lab\_4\_

{

class A

{

public A()

{

Console.WriteLine("constr A()");

this.a = 1;

}

~A()

{

Console.WriteLine(" destr A()");

Thread.Sleep(80000);

}

public virtual int fa() { Console.WriteLine("class A fa()"); return a + 1; }

public int Aa

{

set { Console.Write("set"); }

get { Console.Write("get"); return a; }

}

protected int a = 1;

}

class B : A

{

public B()

{

Console.WriteLine("constr B()");

this.a = 20;

this.b = 10;

this.b1 = -1;

}

~B()

{

Console.WriteLine("destr B()");

}

public override int fa()

{

Console.WriteLine("class B fa()");

base.fa();

a = a + 10;

return a;

}

public int fb()

{

Console.WriteLine("class B fb()");

return a + b + 10;

}

protected int b { set; get; }

public int b1 { set; get; }

}

class K : B

{

public K()

{

Console.WriteLine("constr K()");

this.a = 300;

}

public override int fa() { Console.WriteLine("class K fa()"); return a + 100; }

~K()

{

Console.WriteLine("destr K()");

}

}

abstract class C

{

abstract public int fc();

public void print() { Console.WriteLine("class C print"); }

public int Cc

{

set { Console.Write(""); }

get { Console.Write("get"); return c; }

}

protected int c = 1;

}

class E : C

{

public E() { this.c = 22; }

public override int fc()

{

Console.WriteLine("class E fc()");

return c \* 5;

}

}

class F : C

{

public F() { this.c = 22; }

public override int fc()

{

Console.WriteLine("class F fc()");

return c \* 50;

}

}

interface J

{

int fj\_1();

int fj\_2();

}

class D : J

{

public D() { }

public int fj\_1() { return 250; }

public int fj\_2() { return 500; }

}

class V : K

{

public V() { }

public override int fa() { Console.WriteLine("not use"); return 0; }

}

class Program

{

static void Main(string[] args)

{

A a = new A();

Console.ReadKey();

Console.WriteLine("a.fa()= {0}", a.fa());

a = new B();

Console.ReadKey();

Console.WriteLine("a.fa()= {0}", a.fa());

/\*a = new K();

Console.ReadKey();

Console.WriteLine("a.fa()= {0}", a.fa());\*/

if (a.GetType() == typeof(B))

Console.WriteLine("(a.GetType() == typeof(B))");

else

{

Console.WriteLine("(a.GetType() != typeof(B))");

if (a.GetType() == typeof(K))

Console.WriteLine("(a.GetType() == typeof(K))");

else

Console.WriteLine("(a.GetType() != typeof(K))");

}

{

// K k = new K();

}

//Thread.Sleep(10000);

Console.ReadKey();

Console.WriteLine("Expansion");

Console.WriteLine("((B)a).fb () = {0}", ((B)a).fb()); // Расширение по функции

Console.WriteLine("((B)a).b1 () = {0}", ((B)a).b1); // Расширение по аргументу

Console.ReadKey();

Console.WriteLine("Specification : abstract class");

C c = null;

c = new E();

c.Cc = 455;

c.print();

Console.WriteLine("c.fc()={0}", c.fc());

Console.ReadKey();

c = new F();

c.Cc = 455;

c.print();

Console.WriteLine("c.fc()={0}", c.fc());

Console.ReadKey();

Console.WriteLine("Specification : interface");

J j = null;

j = new D();

Console.WriteLine("j.fj\_1() ={0}", j.fj\_1());

Console.WriteLine("j.fj\_2() ={0}", j.fj\_2());

Console.ReadKey();

Console.WriteLine("Specialization");

a = new K();

Console.WriteLine("a.fa() = {0}", a.fa());

Console.ReadKey();

Console.WriteLine("Construction");

a = new V();

Console.WriteLine(" a.fa= {0}", a.fa());

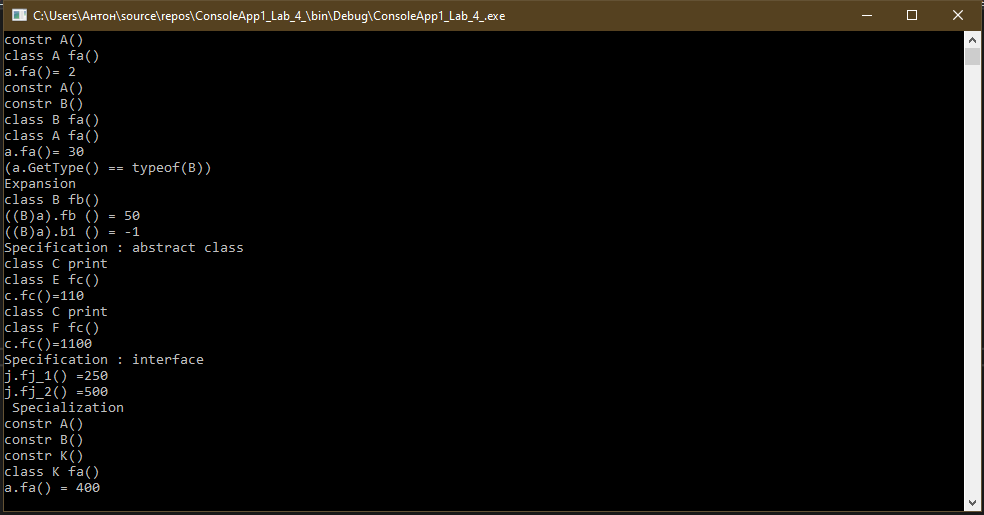
Console.ReadKey();

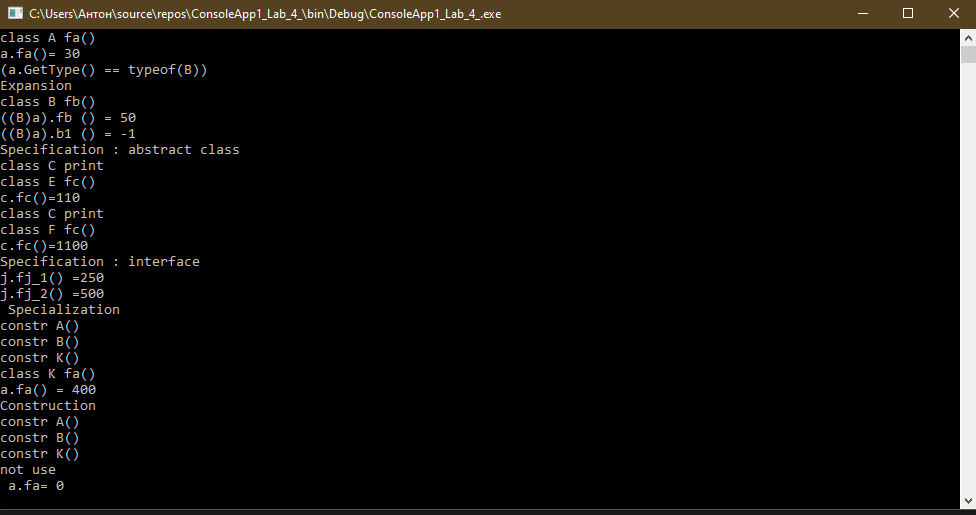
}

}

}

**Результат работы :**

****

****

**Вывод:** Наследование может совершаться несколькими способами. **При расширение** добавляются методы и атрибуты в класс потомка. **При спецификации** создаются абстрактный класс, в котором определяются замещаемые в классе потомка функции, операции, атрибуты и атрибуты доступа, и интерфейс, в котором определяются замещаемые функции. **При специализации** появляется возможность использования данных и функциональности суперкласса в классе потомка. **Конструирование** позволяет в классе потомка изменять реализацию замещаемого метода суперкласса.

**Наследование: комбинирование.**

**А**

**B**

**C**

**J**

**K**

**E**

**F**

public

public

public

public

public

public

public

private

private

private

private

private

private

private

*Рис 2. Наследование: комбинирование.*

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Cryptography.X509Certificates;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_4.\_2\_

{

class K

{

public K() { Console.WriteLine("constr K"); }

public int k { get; set; }

public virtual void fk() { Console.WriteLine("fk K"); }

}

class B:K

{

public B() { Console.WriteLine("constr B"); }

public int b { get; set; }

public virtual void fb() { Console.WriteLine("fb B"); }

public override void fk() { Console.WriteLine("Override fk B"); }

}

interface E

{

double fe();

}

interface F

{

double ff();

}

interface C:E,F

{

int fc();

}

interface J

{

int fj();

int fj1();

}

class A : B, C, J

{

public A() { Console.WriteLine("constr A"); }

public int a { get; set; }

public override void fb()

{

base.fb();

Console.WriteLine(" fb Override A");

Console.WriteLine(" Override fb ");

}

public override void fk() { Console.WriteLine("Override fk A"); }

public double fe() { return 56.25; }

public double ff() { return 112.5; }

public int fc() { return 255; }

public int fj() { return 510; }

public int fj1() { return 1020; }

}

class Program

{

static void Main(string[] args)

{

B b = new B();

b.b = 11;

b.fb();

b.fk();

Console.WriteLine("b.b={0}", b.b);

b = new A();

Console.ReadKey();

b.fb();

b.fk();

Console.ReadKey();

Console.WriteLine("Step 2 : Interface C:E,F ");

C c = null;

c = new A();

Console.WriteLine("c.fc()={0}", c.fc());

Console.WriteLine("c.fe()={0}, c.ff()={1}", c.fe(), c.ff());

Console.WriteLine("Step 3 : Interface J ");

J j = null;

j = new A();

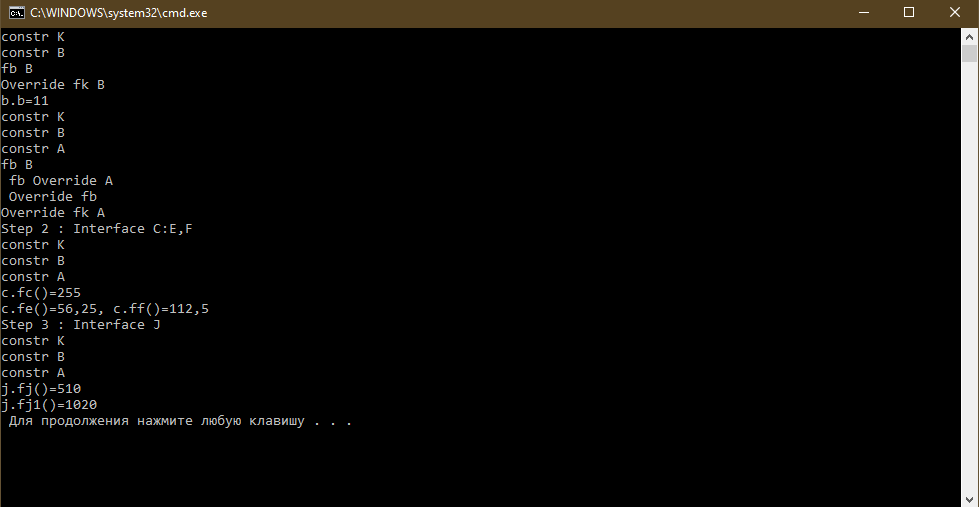
Console.WriteLine("j.fj()={0}", j.fj());

Console.WriteLine("j.fj1()={0}", j.fj1());

Console.ReadKey();

}

}

**Результат работы: **

**Вывод: Комбинирование** позволяет объединить черты нескольких классов и интерфейсов в одном дочернем классе. В программе реализовано наследование класса A от интерфейсов *E,F,J,C* и классов K,B, причем интерфейс C наследуется от интерфейсов E,F, а класс B наследуется от класса K . Происходит множественное наследование, причем для каждого суперкласса можно определить свой вариант наследования, но если в наследуемых классах есть одинаковые методы и/или переменные неоднозначность использования определяется указанием пространства имен класса, переменную или метод которого нужно использовать.

**Лабораторная работа №5. Наследование: комбинирование через общих предков.**

**А**

**B**

**C**

**K**

public

public

public

public

private

private

private

private

*Рис 1. Наследование: комбинирование через общих предков.*

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_5

{

interface K

{

int fk();

}

interface C:K

{

int fc();

int fc\_1();

}

class B:K

{

public B() { Console.WriteLine("Constr B"); }

public int fk() { return 2020; }

public virtual int print()

{

Console.WriteLine("Hello !");

return 0;

}

protected int b {set;get;}

}

class A : B,C

{

public A() { this.a = 1; Console.WriteLine("Constr A"); }

public int fc() { return a + 2020; }

public int fc\_1() { return 2020 - a; }

public int fk() { return a; }

public override int print()

{

base.print();

Console.WriteLine("Hello, I'm here !");

return 0;

}

public int a { set; get; }

}

class Program

{

static void Main(string[] args)

{

K k = null;

k = new A();

Console.WriteLine("k.fk()={0}", k.fk()) ;

Console.WriteLine("((C)k).fc()={0}", ((A)k).fc());

Console.WriteLine("((C)k).fc\_1()={0}", ((A)k).fc\_1());

((A)k).print();

A a = new A();

a.print();

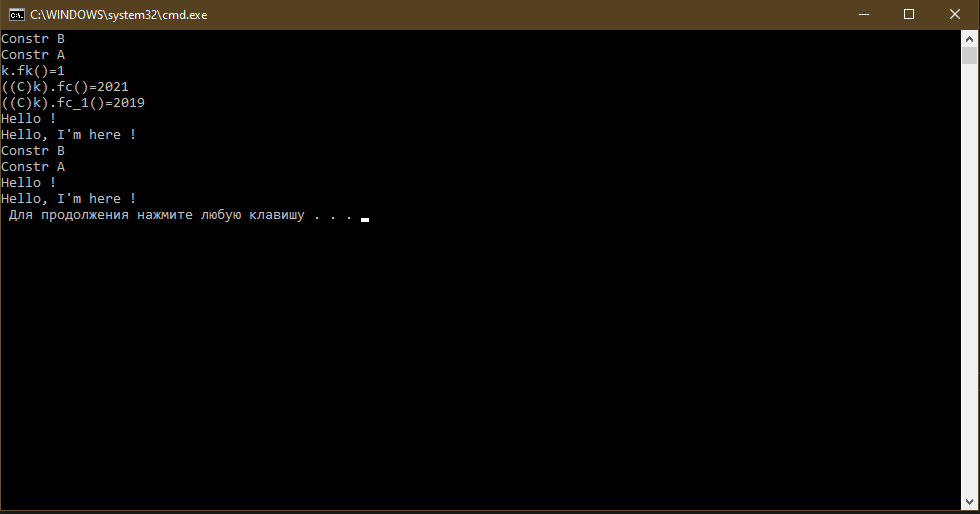
Console.ReadKey();

}

}

}

**Результат программы:**

****

**Вывод:** комбинирование через общих предков позволяет реализовать множественное наследование через общий суперкласс; комбинирует в себе два класса с общим суперклассом, который, в свою очередь, сам присутствует только в единственном экземпляре благодаря virtual наследованию, позволяющее избежать дублирование этого суперкласса.

**Лабораторная работа №6. Ассоциация (один к одному, один ко многим).**

**Один к одному.**

**K**

**E**

public

protected

private

public

private

protected

**1:1**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_6

{

class K

{

public K(){ Console.WriteLine("Constr K (dummy) "); }

public K(E e) { this.e = e; Console.WriteLine("Constr K "); }

public int fk() { return 202; }

~K () { Console.WriteLine("Destr K "); }

public E e = null;

}

class E

{

public E() { Console.WriteLine("Constr E (dummy) "); }

public E(K k) { this.k = k; Console.WriteLine("Constr E "); }

public int fe() { return 404; }

~E() { Console.WriteLine("Destr E "); }

public K k = null;

}

class Program

{

static void Main(string[] args)

{

E e = new E();

K k = new K (e);

e.k = k;

Console.WriteLine("e.k.fk={0}", e.k.fk());

Console.WriteLine("k.e.fe={0}", k.e.fe());

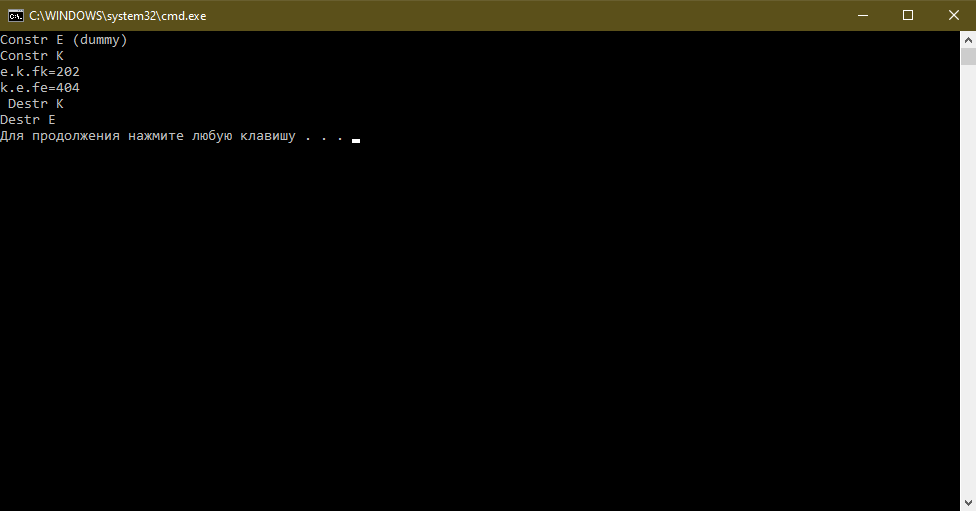
Console.ReadKey();

}

}

}

**Результаты программы**

****

**Вывод:** Ассоциация - двусторонняя зависимость. При ассоциации взаимодействующие объекты, между собой условно находятся на одном уровне. Ни один из объектов не имеет превосходства над другим, и из любого можно получить другой, связанный с ним объект.

**Один ко многим.**

**K**

**E**

public

protected

private

public

private

protected

**1:N**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_6\_1\_N

{

class K

{

public K() { this.e = new E[N]; Console.WriteLine("Constr K "); }

public int fk() { return 202; }

~K() { Console.WriteLine("Destr K "); }

private E[] e = null;

public int size = 0;

private int N = 5;

// 1 способ

public void setE(E e)

{

if (size < N)

{

this.e[size] = e;

size++;

}

}

// 2 cпособ

public E eA

{

set

{

if (size < N) { this.e[size] = value; Console.WriteLine(value.ToString()); size++; }

else Console.WriteLine(" size >= N");

}

get { return this.e[--size]; }

}

}

class E

{

public E() { Console.WriteLine("Constr E (dummy) "); }

public E(K k) { this.k = k; Console.WriteLine("Constr E "); }

public int fe() { return 404; }

~E() { Console.WriteLine("Destr E "); }

public K k { set; get; }

}

class Program

{

static void Main(string[] args)

{

K k = new K();

E e1, e2, e3, e4, e5 = null;

e1 = new E();

e2 = new E();

e3 = new E();

e4 = new E();

e5 = new E();

k.eA = e1;

k.eA = e2;

k.eA = e3;

k.eA = e4;

k.eA = e5;

Console.WriteLine(" k.eA.fe()={0}", k.eA.fe());

Console.WriteLine("size={0}", k.size);

e1.k = k;

e2.k = k;

e3.k = k;

e4.k = k;

e5.k = k;

Console.WriteLine(" e1.k.fk()={0}", e1.k.fk());

Console.WriteLine(" e1.k.fk()={0}", e2.k.fk());

Console.WriteLine(" e1.k.fk()={0}", e3.k.fk());

Console.WriteLine(" e1.k.fk()={0}", e4.k.fk());

Console.WriteLine(" e1.k.fk()={0}", e5.k.fk());

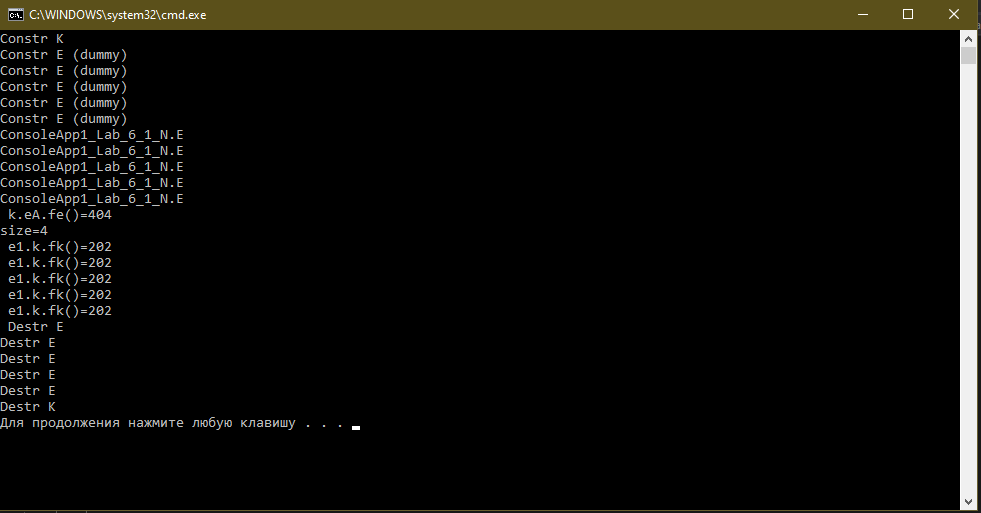
Console.ReadKey();

}

}

}

**Результаты программы:**

****

**Вывод:** В ассоциации 1 : N каждый из N объектов указывает на 1, общий для всех, объект, который в свою очередь имеет N указателей на другие объекты. При этом, из одного объекта группы, с помощью общего можно перейти к другим объектам этой группы.

**Лабораторная работа №7. Использование.**

**K**

**E**

public

protected

private

public

private

protected

**Клиент**

**Сервер**

**Use**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_7

{

class K

{

public K()

{

Console.WriteLine("Constr K");

}

public void fk (E e)

{

Console.WriteLine("algorithm");

this.k = e.e + e.fe();

Console.WriteLine("Es.fe()={0}", Es.fe()); // утилита

Console.WriteLine("Es.fetry()={0}", Es.fetry()); //утилита

}

private int k { set; get; }

}

class E

{

public E()

{ Console.WriteLine("Constr E"); }

public int fe() { return 888; }

public static int fs() { return 2222; }

public int e { set; get; }

}

class Es

{

public static int fe () { return 888; }

public static int fetry () { return 11111; }

}

class Program

{

static void Main(string[] args)

{

K k = new K(); //клиент

E e = new E(); //сервер

k.fk(e);

Console.ReadKey();

Console.WriteLine(" static");

Console.WriteLine("Es.fe()={0}", Es.fe());

Console.WriteLine("Es.fetry()={0}", Es.fetry());

Console.WriteLine("E.fs()={0}", E.fs());

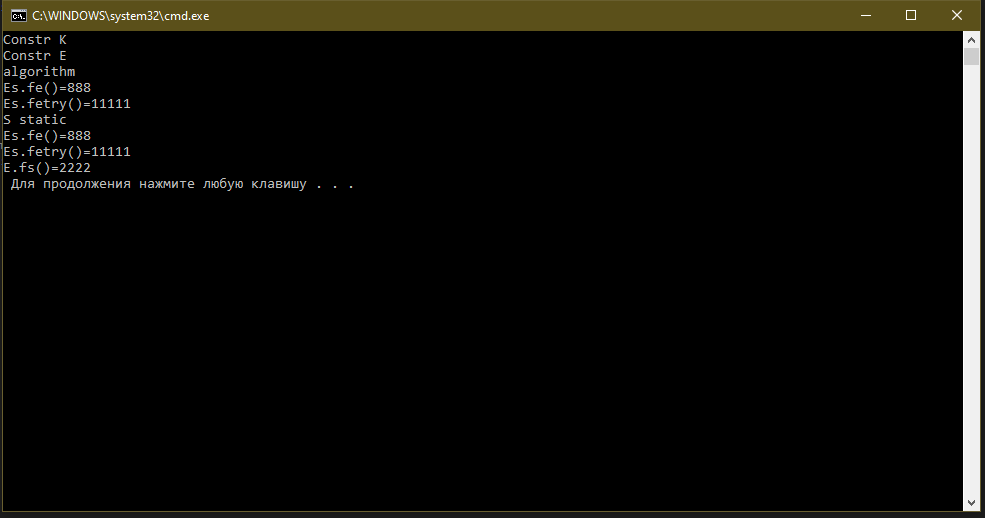
Console.ReadKey();

}

}

}

**Результаты программы:**

****

**Вывод:** С помощью использования клиент-сервер, можно контролировать или управлять одним объектом – клиентом, через другой объект – сервер. При это объект-клиент выступает в качестве параметра для объекта-сервера. Клиент и сервер могут находиться как на одном компьютере (ПК), так и на разных ПК в сети. Также может возникать такая ситуация, когда некоторый программный блок будет одновременно выполнять функции сервера по отношению к одному блоку и клиента по отношению к другому.

**Лабораторная работа №8. Конкретизация: параметров функции, метода, конструктора и атрибутов. Множественная конкретизация. Конкретизация с ограничениями (оператор Is).**

**Swap <T>**

**T**

public

protected

private

public

private

protected

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Expressions;

using System.Management.Instrumentation;

using System.Security.Cryptography.X509Certificates;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_8

{

class Swap

{

public void Fswap(ref A x, ref A y)

{

A t = x;

x = y;

y = t;

}

}

class SwapT <T> where T : class // конкретизация класса -> ,атрибутов, методов, конструкторов.

{

public void Fswap(ref T x, ref T y)

{

T t = x;

x = y;

y = t;

}

}

public interface IA

{

void F();

}

class D : IA

{

public void F() { Console.WriteLine("Class D F()- bonjour ! "); }

}

class U <T> where T : class

{

public U(T t)

{

if (!(t is IA)) Console.WriteLine(" not IA"); // проверка на конструкторе

else Console.WriteLine(" is IA");

this.t = t;

}

public void F1() { Console.WriteLine("Hello"); }

public T t { set; get; }

public void F()

{

Console.WriteLine(t is IA);

Console.WriteLine($"t adress={t.GetHashCode().ToString()}");

if(t is IA)

{

IA ia = (IA)t;

ia.F();

}

else Console.WriteLine("false");

}

}

class L<T1, T2> where T1 : class where T2: class

{

public void f1(T1 t1, T2 t2) { Console.WriteLine("Hi, it's class L and function f1"); }

}

class A { }

class B { }

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Step 1 Swap");

Swap swap = new Swap();

A a1, a2 = null;

a1 = new A();

a2= new A();

Console.WriteLine($"a1 adress={a1.GetHashCode().ToString()}");

Console.WriteLine($"a2 adress={a2.GetHashCode().ToString()}");

Console.WriteLine("");

Console.ReadKey();

swap.Fswap(ref a1, ref a2);

Console.WriteLine($"a1 adress={a1.GetHashCode().ToString()}");

Console.WriteLine($"a2 adress={a2.GetHashCode().ToString()}");

Console.ReadKey();

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("");

Console.WriteLine("Step 2 SwapT");

SwapT<A> swapt = new SwapT<A>();

swapt.Fswap(ref a1, ref a2);

Console.WriteLine($"a1 adress={a1.GetHashCode().ToString()}");

Console.WriteLine($"a2 adress={a2.GetHashCode().ToString()}");

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("");

Console.WriteLine("Step 3 "); //множественная конкретизация

U<A> ua1 = new U<A>(a1);

ua1.F();

D d1 = new D();

Console.WriteLine($" d1 adress={d1.GetHashCode().ToString()}");

U<D> ud1 = new U<D>(d1);

ud1.F();

Console.WriteLine("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

Console.WriteLine("");

Console.WriteLine("Step 4 ");

L<D, A> l = new L<D,A>();

l.f1(d1, a1);

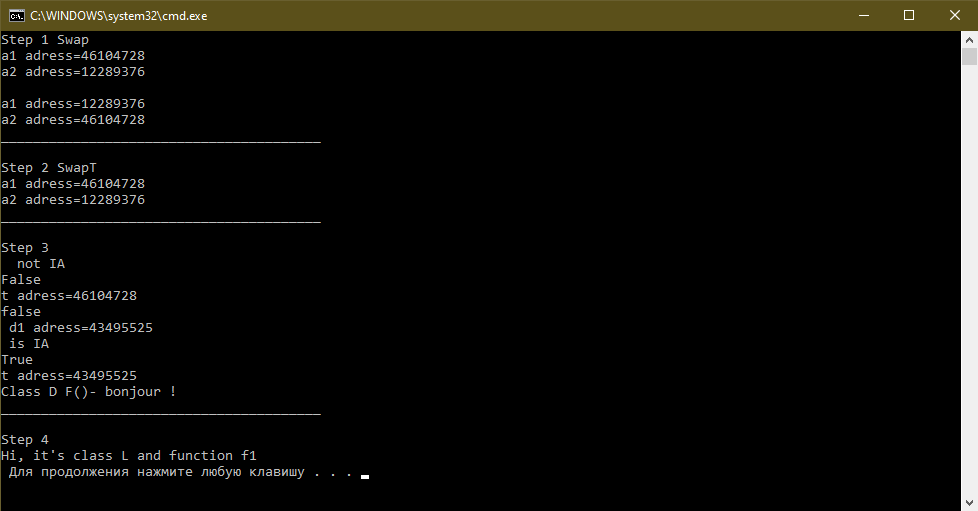
Console.ReadKey();

}

}

}

**Результаты программы:**

****

**Вывод:** Конкретизация позволяет описывать классы и функции для работы с объектами, типы которых заранее не определены. С каким(и) именно типам(и) придется работать определяется в момент вызова – имя типа настраивает функцию\класс для работы.

**Лабораторная работа №9. АНОНИМНЫЕ ФУНКЦИИ: сигнатура функции (delegate), ЛЯМБДА ВЫРАЖЕНИЕ(=>). Event**

**Сигнатура функции (delegate).**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Runtime.CompilerServices;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_9

{

delegate int A(int x, int y);

delegate void B();

class Program

{

static void Main(string[] args)

{

B b = delegate () { Console.WriteLine("B b= delegate () "); Console.ReadKey(); };

Console.WriteLine("A a\_1 = new A (Add)");

A a\_1 = new A(Add);// 2 способ

Console.WriteLine("a\_1(55,5) = {0}", a\_1(13, 14));

A a = null; // 1 cпособ

Console.WriteLine("a= Add");

a = Add;

int result = a(5, 8);

Console.WriteLine("use meth Add: {0}", result);

b();

Console.WriteLine("");

Console.WriteLine("a= Sub");

a = Sub;

Console.WriteLine("use meth Sub a(7,4): {0}", a(7, 4));

b();

Console.WriteLine("");

Console.WriteLine("a= delegate (int x, int y)");

a = delegate (int x, int y) { return x % y; };

Console.WriteLine("use meth delegate (int x, int y) (10,8): {0}", a(10, 8));

b();

Console.WriteLine("");

Console.WriteLine("a= delegate (int a, int b)");

a = delegate (int a1, int b1) { return a1 \* b1; };

Console.WriteLine("use meth delegate (int a1, int b1) (42,20): {0}", a(42, 20));

b();

//////////////////////////////////////////////////////////////////////////////////

Console.WriteLine("");

Console.WriteLine("Step 2");

A apoiner = null;

apoiner = Add;

apoiner += Sub;

apoiner += Dev;

apoiner.Invoke(55, 5);

Console.WriteLine("");

Console.WriteLine("Step 2.1 Summar apoiner {0}" , apoiner.Invoke(55, 5));

apoiner -= Dev;

Console.WriteLine("Step 2.2 apoiner-=Dev {0}", apoiner.Invoke(55, 5));

apoiner += Dev;

apoiner -= Sub;

Console.WriteLine("Step 2.3 apoiner+=Dev;\t apoiner-=Sub\t {0}", apoiner.Invoke(55, 5));

apoiner += delegate (int a1, int b1) { return a1 \* b1; };

Console.WriteLine("Step 2.4 apoiner += delegate (int a1, int b1) {0}", apoiner.Invoke(55, 5));

Console.WriteLine("");

Console.WriteLine("Step 3 Multicast (delegate as parametr)");

Multicast(55, 5, apoiner);

Console.ReadKey();

}

private static int Add(int x1, int y1) { Console.WriteLine("Add {0}", x1 + y1); return x1 + y1; }

private static int Sub(int x2, int y2) { Console.WriteLine("Sub {0}", x2 - y2); return x2 - y2; }

private static int Dev(int x3, int y3) { Console.WriteLine("Dev {0}", x3/ y3); return x3/y3; }

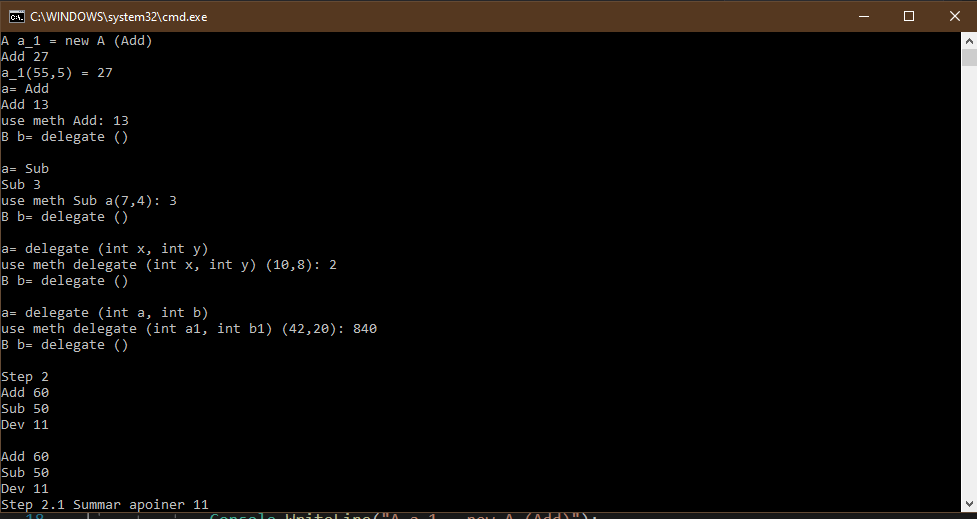
private static void Multicast(int x, int y, A apoiner) { apoiner.Invoke(x, y); }

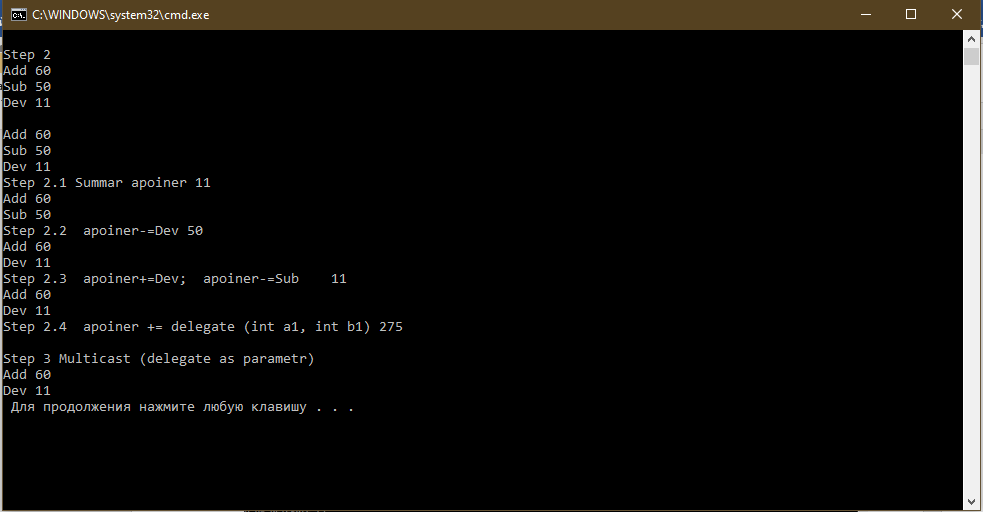
// private static int Sub(int x2, int y2) { return x2 + y2; }

}

}

**Результат программы:**

****

****

**Вывод:** Для определения делегата используется ключевое слово delegate, после которого идет сигнатура. Для использования делегата объявляется объект этого делегата. Объекту данного класса сигнатур можно присвоить метод со сходной сигнатурой. Для объектов класса сигнатур возможна операция декремента/инкремента. Также можно использовать анонимные делегаты путём объявления объекта класса сигнатур и присваивания ему анонимного метода со сходной у делегата сигнатурой. Делегаты можно передавать в качестве аргументов методу, что довольно удобно. Использование делегатов позволяет добиться более адекватного моделирования и повторного использования кода.

**ЛЯМБДА ВЫРАЖЕНИЕ(=>).**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_9\_Lam

{

delegate int Lambda(int x, int y);

delegate void Lambda\_OP();

class Program

{

static void Main(string[] args)

{

Console.WriteLine(" step 1 Lambda");

//Lambda lambda = null;

Lambda lambda = (x, y) => { return x + y; };

Console.WriteLine($"lambda(55,5)= {lambda.Invoke(55, 5)}");

Lambda\_OP print = () => Console.WriteLine("hello, it's Lambda\_OP print ");

print();

Console.WriteLine(" step 2 Using Lambda");

Func<int, int> square = s => s \* s;

Console.WriteLine($"square = {square (81)}");

Func<int, double> radical= r => Math.Sqrt(r);

Console.WriteLine($"radical = {radical(81)}");

Console.WriteLine(" Find index of first \_Nechet\_ element in list ");

List<int> elements = new List<int>() { 0, 1, 1, 2, 3, 5, 8 };

int NechetIndex = elements.FindIndex(x => x % 2 != 0);

Console.WriteLine($"NechetIndex = { NechetIndex}");

int NechetChislo = elements.Find(x => x % 2 != 0);

Console.WriteLine($"NechetChislo = { NechetChislo}");

int Chislo\_List = elements.Find(x => (x >= 4) && (x <= 8));

Console.WriteLine($" Chislo\_List [4;8] = { Chislo\_List}");

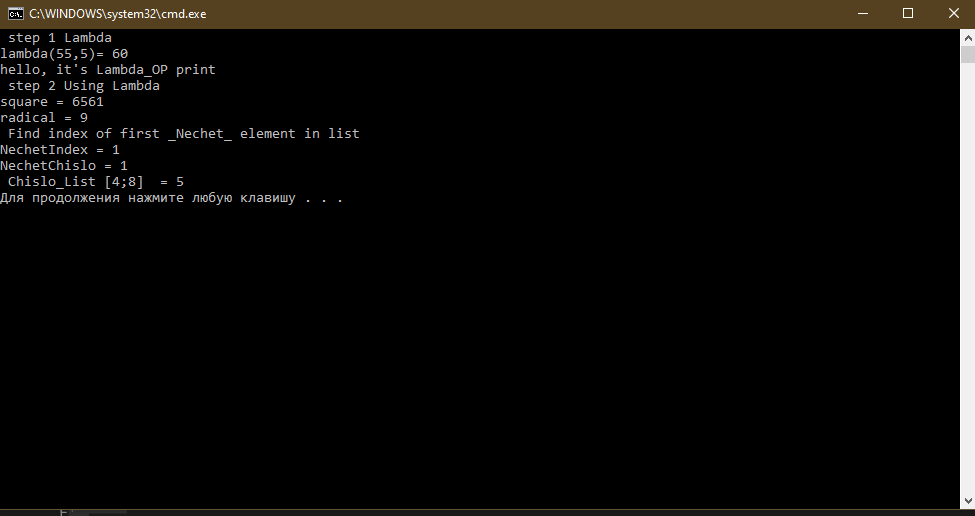
Console.ReadKey();

}

}

}

**Результат программы:**

****

**Вывод:** Лямбда-выражения представляют упрощенную запись анонимных методов. Лямбда-выражения позволяют создать емкие лаконичные методы, которые могут возвращать некоторое значение и которые можно передать в качестве параметров в другие методы. Ламбда-выражения имеют следующий синтаксис: слева определяется список параметров, в середине лямбда-оператор =>, а справа блок выражений, использующий эти параметры: (список\_параметров) => выражение. Как и делегаты, лямбда-выражения можно передавать в качестве аргументов методу для тех параметров, которые представляют делегат, что довольно удобно. Как и делегаты, лямбда-выражения позволяют добиться повторного использования кода и адекватного моделирования.

**Event.**

**Текст программы**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Net.Http;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApp1\_Lab\_9\_Ev

{

public class Publisher

{

public delegate void PublisherEventHandler(Message message);

public event PublisherEventHandler changed;

public Publisher() { }

public void EventForPublisher(Message message) {changed(message); Console.WriteLine("Event for all subscribers= {0}", message.message); }

}

public class Message:EventArgs

{

public Message(string message): base() { this.message = message; }

public string message { set; get; }

}

public class Subscribers

{

public Subscribers (int QRC) { this.QRC = QRC; }

public int QRC{ set; get; }

public void subscribe(Message message) { Console.WriteLine("subscriber= {0}, {1}", this.QRC, message.message); }

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("step 1 Create");

Publisher publisher = new Publisher();

Subscribers Subscriber1 = new Subscribers(1);

Subscribers Subscriber2 = new Subscribers(2);

Subscribers Subscribern = new Subscribers(999);

Console.WriteLine("step 2 Sub");

publisher.changed += Subscriber1.subscribe;

publisher.changed += Subscriber2.subscribe;

publisher.changed += Subscribern.subscribe;

Console.WriteLine("step 3 Event");

publisher.EventForPublisher(new Message("New book is ready!"));

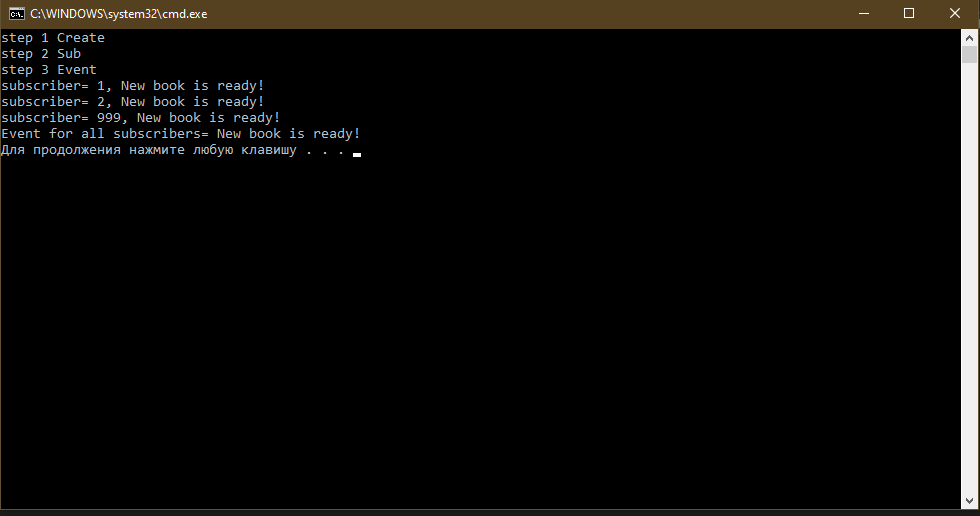
Console.ReadKey();

}

}

}

**Результат программы:**

****

**Вывод:** События позволяют сигнализируют системе о том, что произошло определенное действие. События объявляются в классе с помощью ключевого слова event, после которого идет название класса сигнатур, для которого объявляется это событие. Обработчик события – это метод, который выполняет некоторые действия в программе, в случае если событие произошло(сгенерировалось).