**Министерство образования и науки Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**Псковский государственный университет**

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

**Контрольная работа № 1**

по учебной дисциплине

**Объектно-ориентированное программирование (С#)**

Выполнил студент

группы 1022-03

Ковалевский Р.А.

Проверил: Антонов И.Н.

Дата проверки:\_\_\_\_\_\_\_\_

**Псков**

**2018**

**Классы**

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В C# используется спецификация класса для построения объектов, которые являются экземплярами класса. Следовательно, класс, по существу, представляет собой ряд схематических описаний способа построения объекта. При этом очень важно подчеркнуть, что класс является логической абстракцией. Физическое представление класса появится в оперативной памяти лишь после того, как будет создан объект этого класса.

Классы и структуры — это, по сути, шаблоны, по которым можно создавать объекты. Каждый объект содержит данные и методы, манипулирующие этими данными.

Общая форма определения класса

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое.

Вообще говоря, данные содержатся в членах данных, определяемых классом, а код — в функциях-членах. Следует сразу же подчеркнуть, что в C# предусмотрено несколько разновидностей членов данных и функций-членов. (Приложение 1.)

Данные-члены

Данные-члены — это те члены, которые содержат данные класса — поля, константы, события. Данные-члены могут быть статическими (static). Член класса является членом экземпляра, если только он не объявлен явно как static.

Давайте рассмотрим виды этих данных:

1.Поля (field)

2.Это любые переменные, ассоциированные с классом.

3.Константы

Константы могут быть ассоциированы с классом тем же способом, что и переменные. Константа объявляется с помощью ключевого слова const. Если она объявлена как public, то в этом случае становится доступной извне класса.

События

Это члены класса, позволяющие объекту уведомлять вызывающий код о том, что случилось нечто достойное упоминания, например, изменение свойства класса либо некоторое взаимодействие с пользователем. Клиент может иметь код, известный как обработчик событий, реагирующий на них.

Функции-члены

Функции-члены — это члены, которые обеспечивают некоторую функциональность для манипулирования данными класса. Они включают методы, свойства, конструкторы, финализаторы, операции и индексаторы:

Методы (method)

Это функции, ассоциированные с определенным классом. Как и данные-члены, по умолчанию они являются членами экземпляра. Они могут быть объявлены статическими с помощью модификатора static.

Свойства (property)

Это наборы функций, которые могут быть доступны клиенту таким же способом, как общедоступные поля класса. В C# предусмотрен специальный синтаксис для реализации чтения и записи свойств для классов, поэтому писать собственные методы с именами, начинающимися на Set и Get, не понадобится. Поскольку не существует какого-то отдельного синтаксиса для свойств, который отличал бы их от нормальных функций, создается иллюзия объектов как реальных сущностей, предоставляемых клиентскому коду.

Конструкторы (constructor)

Это специальные функции, вызываемые автоматически при инициализации объекта. Их имена совпадают с именами классов, которым они принадлежат, и они не имеют типа возврата. Конструкторы полезны для инициализации полей класса.

Финализаторы (finalizer)

Вызываются, когда среда CLR определяет, что объект больше не нужен. Они имеют то же имя, что и класс, но с предшествующим символом тильды. Предсказать точно, когда будет вызван финализатор, невозможно.

Операции (operator)

Это простейшие действия вроде + или -. Когда вы складываете два целых числа, то, строго говоря, применяете операцию + к целым. Однако C# позволяет указать, как существующие операции будут работать с пользовательскими классами (так называемая перегрузка операции).

Индексаторы (indexer)

Позволяют индексировать объекты таким же способом, как массив или коллекцию.

Класс создается с помощью ключевого слова class. Ниже приведена общая форма определения простого класса, содержащая только переменные экземпляра и методы:

class имя\_класса {

// Объявление переменных экземпляра.

доступ тип переменная1;

доступ тип переменная2;

//...

доступ тип переменнаяN;

// Объявление методов.

доступ возращаемый\_тип метод1 (параметры) {

// тело метода

}

доступ возращаемый\_тип метод2 (параметры) {

// тело метода

}

//. . .

доступ возращаемый\_тип методN(параметры) {

// тело метода

}

}

Обратите внимание на то, что перед каждым объявлением переменной и метода указывается доступ. Это спецификатор доступа, например public, определяющий порядок доступа к данному члену класса. Члены класса могут быть как закрытыми (private) в пределах класса, так открытыми (public), т.е. более доступными. Спецификатор доступа определяет тип разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса.

Давайте разберем пример создания класса, описывающего характеристики пользователя:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class UserInfo

{

// Поля класса

public string Name, Family, Adress;

public byte Age;

// Метод, выводящий в консоль контактную информацию

public void writeInConsoleInfo(string name, string family, string adress, byte age)

{

Console.WriteLine("Имя: {0}\nФамилия: {1}\nМестонахождение: {2}\nВозраст: {3}\n", name, family, adress, age);

}

}

class Program

{

static void Main(string[] args)

{

// Создаем объект типа UserInfo

UserInfo myInfo = new UserInfo();

myInfo.Name = "Alexandr";

myInfo.Family = "Erohin";

myInfo.Adress = "ViceCity";

myInfo.Age = 26;

// Создадим новый экземпляр класса UserInfo

UserInfo myGirlFriendInfo = new UserInfo();

myGirlFriendInfo.Name = "Elena";

myGirlFriendInfo.Family = "Korneeva";

myGirlFriendInfo.Adress = "ViceCity";

myGirlFriendInfo.Age = 22;

// Выведем информацию в консоль

myInfo.writeInConsoleInfo(myInfo.Name, myInfo.Family, myInfo.Adress, myInfo.Age);

myGirlFriendInfo.writeInConsoleInfo(myGirlFriendInfo.Name,myGirlFriendInfo.Family,myGirlFriendInfo.Adress,myGirlFriendInfo.Age);

Console.ReadLine();

}

**Инкапсуляция**

Инкапсуляция — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, полиморфизмом и наследованием).

Инкапсуля́ция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента (что у него внутри?), а взаимодействовать с ним посредством предоставляемого интерфейса (публичных методов и членов), а также объединить и защитить жизненно важные для компонента данные. При этом пользователю предоставляется только спецификация (интерфейс) объекта.

Пользователь может взаимодействовать с объектом только через этот интерфейс. Реализуется с помощью ключевого слова: public.

Не пытайтесь найти перевод этого термина с английского — вы получите encapsulation. Зато если вы попробуйте присмотреться, вы увидите главное — «капсула». Каждый объект не должен выставлять наружу все свои параметры для изменения просто так. Например, если у нашего робота есть координаты X и Y, то не вызывает сомнений факт, что их нельзя менять прямо. Робот должен поменять свои координаты в результате передвижения. Нельзя обратиться к переменной X внутри объекта Robot и сделать самое простое присваивание. Это будет как минимум нелогично. Лучше такого вообще не позволять. Т.е. мы таким образом должны создавать описание класса, чтобы нельзя было просто так получать доступ к его внутренним переменным. Это будет похоже на то, как если бы мы при производстве телевизора давали людям доступ ко всей схеме и каждый мог переключать проводки внутри него напрямую. Обычный человек таким телевизором вряд ли бы пользовался. Конечно нашлось бы несколько энтузиастов, которые обрадовались такому положению дел. Но это скорее всего экзотика. Более правильно в случае с роботом было сделать так, чтобы можно было получить значения координат. А менять координаты можно было бы только в результате движения робота. Проехал он 10 метров — поменялись его координаты X и Y в соответствии с расстоянием и курсом. Причем автоматически, т.е. закрытие внутренних переменных — хорошая идея. Для правильного поведения объекта. Нам самим будет проще — при условии, что класс более-менее разумно спроектирован и реализован. Но возникает резонный вопрос — а как тогда обращаться к внутренним переменным. В какой-то момент мы будем вынуждены это сделать. Об этом мы узнаем прямо сейчас. А заодно посмотрим на класс. Перед нами достаточно простое описание робота. Рассмотрим его внимательно.

public class Robot

{// Текущая координата X

private double x = 0;

// Текущая координата Y

private double y = 0;

// Текущий курс (в градусах)

private double course = 0;

// Передвижение на дистанцию distance

public void forward(int distance) {

// Обращение к полю объекта X

x = x + distance \* Math.cos(course / 180 \* Math.PI);

// Обращение к полю объекта Y

y = y + distance \* Math.sin(course / 180 \* Math.PI);

} // Печать координат робота

public void printCoordinates() {

System.out.println(x + "," + y);

} public double getX() {

return x;

}

public double getY() {

return y;

}

public double getCourse() {

return course;

}

// Мы рассмотрим этот метод несколько позже

public void setCourse(double course) {

this.course = course;

В этом примере есть несколько моментов, на которые надо обратить внимание.  
Первый — возле объявления внутренних переменных появилось ключевое слово private. Это слово определяет видимость переменной (и не только — мы это увидим). Что значит «видимость»? По сути это означает в какой части кода программы можно будет обратиться к этой переменной.  
Всего существует три варианта таких слов:

1. private
2. protected
3. public

Слово «private» дает самую слабую видимость — только внутри описания/методов самого класса. Т.е. в методе forward переменные и course видимы. Но если вы попробуете обратиться к этим перменным так, как мы делали в предыдущей части из класса RobotManager, то ваша программа не скомпилируется. Вам будет указано, что у класса Robot переменная x (или y) — не доступна.

Слово «protected» позволяет видеть переменные другим классам, но не всем. Когда мы доберемся до понятие наследования и пакетов, мы рассмотрим «protected» еще раз. Пока дадим просто описание — переменные будут видны в классах-наследниках, и классах, находящихся в том же пакете. Может пока не очень понятно, но наберитесь терпения.   
И наконец «public». Это описание позволяет обращаться к переменной методу откуда угодно. Вы наверняка увидели public в коде класса Robot при объявлении методов.

Пользователь не может использовать закрытые данные и методы. Реализуется с помощью ключевых слов: private, protected, internal.

**Наследование**

Насле́дование — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с инкапсуляцией, полиморфизмом и абстракцией), позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

Другими словами, класс-наследник реализует спецификацию уже существующего класса (базовый класс). Это позволяет обращаться с объектами класса-наследника точно так же, как с объектами базового класса.

Простое наследование:

Класс, от которого произошло наследование, называется базовым или родительским (англ. base class). Классы, которые произошли от базового, называются потомками, наследниками или производными классами (англ. derived class).

В некоторых языках используются абстрактные классы. Абстрактный класс — это класс, содержащий хотя бы один абстрактный метод, он описан в программе, имеет поля, методы и не может использоваться для непосредственного создания объекта. То есть от абстрактного класса можно только наследовать. Объекты создаются только на основе производных классов, наследованных от абстрактного. Например, абстрактным классом может быть базовый класс «сотрудник вуза», от которого наследуются классы «аспирант», «профессор» и т. д. Так как производные классы имеют общие поля и функции (например, поле «год рождения»), то эти члены класса могут быть описаны в базовом классе. В программе создаются объекты на основе классов «аспирант», «профессор», но нет смысла создавать объект на основе класса «сотрудник вуза».

Множественное наследование :

При множественном наследовании у класса может быть более одного предка. В этом случае класс наследует методы всех предков. Достоинства такого подхода в большей гибкости. Множественное наследование реализовано в C++. Из других языков, предоставляющих эту возможность, можно отметить Python и Эйфель. Множественное наследование поддерживается в языке UML.

Множественное наследование — потенциальный источник ошибок, которые могут возникнуть из-за наличия одинаковых имен методов в предках. В языках, которые позиционируются как наследники C++ (Java, C# и др.), от множественного наследования было решено отказаться в пользу интерфейсов. Практически всегда можно обойтись без использования данного механизма. Однако, если такая необходимость все-таки возникла, то, для разрешения конфликтов использования наследованных методов с одинаковыми именами, возможно, например, применить операцию расширения видимости — «::» — для вызова конкретного метода конкретного родителя.

Попытка решения проблемы наличия одинаковых имен методов в предках была предпринята в языке Эйфель, в котором при описании нового класса необходимо явно указывать импортируемые члены каждого из наследуемых классов и их именование в дочернем классе.

Большинство современных объектно-ориентированных языков программирования (C#, Java, Delphi и др.) поддерживают возможность одновременно наследоваться от класса-предка и реализовать методы нескольких интерфейсов одним и тем же классом. Этот механизм позволяет во многом заменить множественное наследование — методы интерфейсов необходимо переопределять явно, что исключает ошибки при наследовании функциональности одинаковых методов различных классов-предков.

Пример наследования :

\* Родительский класс Human \*/

class Human {

private $name;

/\* Конструктор (в нем мы задаем поле $name при создании экземпляра класса) \*/

public function \_\_construct($name) {

$this->name = $name;

}

/\* Метод say(). Предполагаем, что Human изначально может говорить \*/

public function say() {

echo "Меня зовут ".$this->name." и ";

}

}

/\* Класс Мужчина. Ключевым словом extends мы наследуем родителя Human \*/

class Man extends Human {

public function beard() {

echo "у меня растет борода";

}

}

/\* Класс Женщина. Ключевым словом extends мы наследуем родителя Human \*/

class Women extends Human {

public function bearChildren() {

echo "я рожаю детей";

}

}

/\* Создаем экземпляр класса Man и вызываем методы. \*/

$man = new Man("Sergey");

$man->say();

$man->beard();

/\* Создаем экземпляр класса Women и вызываем методы. \*/

$women = new Women("Maria");

$women->say();

$women->bearChildren();

**Полиморфизм**

Полиморфи́зм— возможность объектов с одинаковой спецификацией иметь различную реализацию. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций». Полиморфизм — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием). Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение: внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов и их количеством); внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

Полиморфи́зм — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования.

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

Полиморфизм — один из четырёх важнейших механизмов объектно-ориентированного программирования (наряду с абстракцией, инкапсуляцией и наследованием).

Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяются в систему, которую могут называть по-разному — интерфейс, класс. Общность имеет внешнее и внутреннее выражение:

* внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов и их количеством);
* внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций.

Формы полиморфизма

Используя Параметрический полиморфизм можно создавать универсальные базовые типы. В случае параметрического полиморфизма, функция реализуется для всех типов одинаково и таким образом функция реализована для произвольного типа. В параметрическом полиморфизме рассматриваются параметрические методы и типы.

Параметрические методы.

 Если полиморфизм включения влияет на наше восприятие объекта, то параметрические полиморфизм влияет на используемые методы, так как можно создавать методы родственных классов, откладывая объявление типов до времени выполнения. Для избежания написания отдельного метода каждого типа применяется параметрический полиморфизм, при этом тип параметров будет являться таким же параметром, как и операнды.

Параметрические типы.

Вместо того, чтобы писать класс для каждого конкретного типа следует создать типы, которые будут реализованы во время выполнения программы то есть мы создаем параметрический тип.

Полиморфизм переопределения.

Абстрактные методы часто относятся к отложенным методам. Класс, в котором определен этот метод может вызвать метод и полиморфизм обеспечивает вызов подходящей версии отложенного метода в дочерних классах. Специальный полиморфизм допускает специальную реализацию для данных каждого типа.

Полиморфизм

Любое обучение вождению не имело бы смысла, если бы человек, научившийся водить, скажем, ВАЗ 2106 не мог потом водить ВАЗ 2110 или BMW X3. С другой стороны, трудно представить человека, который смог бы нормально управлять автомобилем, в котором педаль газа находится левее педали тормоза, а вместо руля-джостик .  
Всё дело в том, что основные элементы управления автомобиля имеют одну и ту же конструкцию и принцип действия. Водитель точно знает, что для того, чтобы повернуть налево, он должен повернуть руль, независимо от того, того, есть там гидроусилитель или нет.

Если человеку надо доехать с работы до дома, то он сядет за руль автомобиля и будет выполнять одни и те же действия, независимо от того, какой именно тип автомобиля он использует. По сути, можно сказать, что все автомобили имеют один и тот же интерфейс, а водитель, абстрагируясь от сущности автомобиля, работает именно с этим интерфейсом. Если водителю предстоит ехать по немецкому автобану, он, вероятно выберет быстрый автомобиль с низкой посадкой, а если предстоит возвращаться из отдалённого маральника в Горном Алтае после дождя, скорее всего, будет выбран УАЗ с армейскими мостами. Но, независимо от того, каким образом будет реализовываться движение и внутреннее функционирование машины, интерфейс останется прежним.

Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Например, если вы читаете данные из файла, то, очевидно, в классе, реализующем файловый поток, будет присутствовать метод похожий на следующий: byte readBytes   
Предположим теперь, что вам необходимо считывать те же данные из сокета. В классе, реализующем сокет, также будет присутствовать метод readBytes. Достаточно заменить в вашей системе объект одного класса на объект другого класса, и результат будет достигнут.   
  
 При этом логика системы может быть реализована независимо от того, будут ли данные прочитаны из файла или получены по сети. Таким образом, мы абстрагируемся от конкретной специализации получения данных и работаем на уровне интерфейса. Единственное требование при этом – чтобы каждый используемый объект имел метод readBytes.

Давайте посмотрим на примере реализацию полиморфизма в PHP:

/\* Это интерфейс Say \*/

interface Say {

public function say();

}

/\* Это абстрактный класс Human имплементирующий интерфейс Say \*/

abstract class Human implements Say{

private $name;

public function \_\_construct($name) {

$this->name = $name;

}

public function getName() {

return $this->name;

}

}

/\* Класс Man наследуют класс Human и обязан реализовать метод say() \*/

class Man extends Human {

public function \_\_construct($name) {

parent::\_\_construct($name);

}

public function beard() {

echo "у меня растет борода";

}

public function say() {

echo "У меня мужской голос, меня зовут ".$this->getName()." и ";

}

}

/\* Класс Women наследуют класс Human и обязан реализовать метод say() \*/

class Women extends Human {

public function \_\_construct($name) {

parent::\_\_construct($name);

}

public function bearChildren() {

echo "я рожаю детей";

}

public function say() {

echo "У меня женский голос, меня зовут ".$this->getName()." и ";

}

}

$man = new Man("Sergey");

$man->say();

$man->beard();

$women = new Women("Maria");

$women->say();

$women->bearChildren();

**Контрольная работа №2**

abstract class Figure

{

protected int X, Y;//информация о начальной точке

public Figure(int X, int Y)//конструктор класса, получающий эту точку в качестве аргументов

{

this.X = X;

this.Y = Y;

}

public abstract void Draw(Graphics g);//рисование

public abstract void DrawLine(Graphics g)//рисование лини

public abstract void DrawDash(Graphics g)//рисование пунктиром

public abstract void Hide(Graphics g);//стирание фигуры

}

// Для фигуры прямоугольник

class Rect : Figure

{

private int x1, x2, y1, y2;

public Rect(int X, int Y, MouseEventArgs e)

: base(X, Y)

{

if (e.X < X)

{

x2 = X; x1 = e.X;

}

else

{

x1 = X; x2 = e.X;

}

if (e.Y < Y)

{

y2 = Y; y1 = e.Y;

}

else

{

y1 = Y; y2 = e.Y;

}

}

public override void DrawLine(Graphics g)// метод рисования линии

{

Pen pen1 = new Pen(Color.Black, 1);

g.DrawLine(x1, y1, x2,y2); // рисует линии

Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);

g.DrawRectangle(pen1, rect);

}

public override void DrawDash(Graphics g)// метод рисования пунктир

{

Pen pen1 = new Pen(Color.Black, 1);

pen1.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;// пунктирной линией

Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);

g.DrawRectangle(pen1, rect);

}

public override void Hide(Graphics g)//стирание

{

Pen pen1 = new Pen(Color.White, 1);

Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);

g.DrawRectangle(pen1, rect);

}

public override void Draw(Graphics g)//рисование

{

Pen pen1 = new Pen(Color.Black, 1);

Rectangle rect = new Rectangle(x1, y1, x2 - x1, y2 - y1);

g.DrawRectangle(pen1, rect);

}

// Для фигуры Ellips

class Ellips: Figure

{

private int x1, x2, y1, y2;

public Ellips(int X, int Y, MouseEventArgs e)

: base(X, Y)

{

if (e.X < X)

{

x2 = X; x1 = e.X;

}

else

{

x1 = X; x2 = e.X;

}

if (e.Y < Y)

{

y2 = Y; y1 = e.Y;

}

else

{

y1 = Y; y2 = e.Y;

}

}

public override void DrawEllipseRectagle(Graphics g)// метод рисования линии

{

Pen pen1 = new Pen(Color.Black, 1);

g.DrawLine(x1, y1, x2,y2); // рисует линии

Ellips rect = new Ellips(x1, y1, x2 - x1, y2 - y1);

g.DrawEllipse(pen1, rect);

}

public override void DrawDash(Graphics g)// метод рисования пунктир

{

Pen pen1 = new Pen(Color.Black, 1);

pen1.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;// пунктирной линией

Ellips rect = new Ellips(x1, y1, x2 - x1, y2 - y1);

g.DrawEllipse(pen1, rect);

}

public override void Hide(Graphics g)//стирание

{

Pen pen1 = new Pen(Color.White, 1);

Ellips rect = new Ellips(x1, y1, x2 - x1, y2 - y1);

g.DrawEllipse(pen1, rect);

}

public override void Draw(Graphics g)//рисование

{

Pen pen1 = new Pen(Color.Black, 1);

Ellips rect = new Ellips(x1, y1, x2 - x1, y2 - y1);

g.DrawEllipse(pen1, rect);

}

// Для Линии

class Line: Figure

{

private int x1, x2, y1, y2;

public Line(int X, int Y, MouseEventArgs e)

: base(X, Y)

{

if (e.X < X)

{

x2 = X; x1 = e.X;

}

else

{

x1 = X; x2 = e.X;

}

if (e.Y < Y)

{

y2 = Y; y1 = e.Y;

}

else

{

y1 = Y; y2 = e.Y;

}

}

public override void DrawLinePoint(Graphics g)// метод рисования линии

{

Pen pen1 = new Pen(Color.Black, 1);

g.DrawLine(x1, y1, x2,y2); // рисует линии

Line rect = new Line(x1, y1, x2 - x1, y2 - y1);

g.DrawLine(pen1, rect);

}

public override void DrawDash(Graphics g)// метод рисования пунктир

{

Pen pen1 = new Pen(Color.Black, 1);

pen1.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;// пунктирной линией

Line rect = new Line(x1, y1, x2 - x1, y2 - y1);

g.DrawLine(pen1, rect);

}

public override void Hide(Graphics g)//стирание

{

Pen pen1 = new Pen(Color.White, 1);

Line rect = new Line(x1, y1, x2 - x1, y2 - y1);

g.DrawLine(pen1, rect);

}

public override void Draw(Graphics g)//рисование

{

Pen pen1 = new Pen(Color.Black, 1);

Line rect = new Line(x1, y1, x2 - x1, y2 - y1);

g.DrawLine(pen1, rect);

}

**Список используемой литературы :**

1. Радченко Г.И, Е.А. Захаров. Объектно-ориентированное программирование. Челябинск: Издательский центр ЮУрГУ, 2013. - 167 с.
2. Основные принципы ООП: инкапсуляция, наследование, полиморфизм :[Электронный ресурс]. URL: <http://gos-it.wikia.com/wiki/>

(Дата обращения :04.03.2018).

1. Васильев, А. Н. Java. Объектно-ориентированное программирование / А.Н. Васильев. - М.: Питер, 2012. - 398 c.
2. Инкапсуляция. StudFiles. :[Электронный ресурс]. URL: https://studfiles.net/preview/1647855/page:2/

(Дата обращения :06.06.2018).

1. Кьоу, Дж. Объектно-ориентированное программирование / Дж. Кьоу, М. Джеанини. - М.: Питер, 2005. - 240 c.