**Лекция 15. Основы объектно-ориентированного программирования**

*Цель лекции:* Формирование у студентов знаний, связанных с механизмами объектно-ориентированного программирования.

В результате изучения данной лекции студенты будут способны:

* Продемонстрировать понимание механизмов объектно-ориентированного программирования
* Продемонстрировать знание синтаксиса работы с классами

Объектно-ориентированное направление программирования (ООП) появилось в конце семидесятых годов XX века из-за необходимости реализации больших проектов, которым уже не удовлетворяли возможности структурного программирования.

Все программы на С# являются объектно-ориентированными.

ООП основано на следующих свойствах: абстракция, инкапсуляция, полиморфизм и наследование.

***Классы, объекты***

**Класс** – разновидность абстрактного типа данных в ООП, характеризуемый способом своего построения. Класс представляет собой шаблон для создания *объектов*. **Объекты** являются *экземплярами* классов. Класс – логическая абстракция, описание, по которому строятся объекты.

Объекты – реальные структуры реализации классов, занимающие после создания оперативную память. Классы можно сравнить с чертежами, по которым можно создать реальные объекты. Для создания класса выделяют логически различимое направление рассматриваемой области (абстракция). Классы, как и получаемые по их описаниям объекты, состоят из объединения членов (инкапсуляция) – данных-полей и кода-методов.

Методы объектов близкой спецификации могут иметь одинаковые имена и назначение, но выполнять действия, характерные для конкретного объекта (полиморфизм). При необходимости получения новых возможностей можно добавить дочернему классу недостающие члены, позаимствовав уже существующие у родительского класса (наследование).

***Абстракция***

Абстракция – это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей её реализации в терминах более примитивных функций. Таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

Такой подход является основой объектно-ориентированного программирования. Это позволяет работать с объектами, не вдаваясь в особенности их реализации. В каждом конкретном случае применяется тот или иной подход: инкапсуляция, полиморфизм или наследование. Например, при необходимости обратиться к скрытым данным объекта, следует воспользоваться инкапсуляцией, создав, так называемую, функцию доступа или свойство.

Правильно сконструированный **класс должен определять только одну логическую сущность.** Например, класс описания технологического оборудования не содержит описание сотрудника и наоборот. В правильно сконструированном классе должна быть сгруппирована логически связанная информация, иначе структурированность кода будет нарушена.

Абстракция выделяет рассматриваемый объект, определяет набор данных, описывающих его и действия, которые можно с ними производить. Класс может иметь модификатор *abstract*, который указывает, что описываемый класс является абстрактным, т.е. не может использоваться для создания экземпляров, а только для создания дочерних классов.

***Инкапсуляция***

**Инкапсуляция** – свойство языка программирования, позволяющее объединить и защитить данные и код в объекте и скрыть реализацию объекта от пользователя (прикладного программиста). При этом пользователю предоставляется только спецификация (интерфейс) объекта.

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

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

Таблица 15.1 Модификаторы доступа, указывающие уровень доступности к членам класса

Объявленная доступность Описание

***public*** Неограниченный доступ

***protected*** Доступ ограничен содержащим классом или типами, которые являются производными от содержащего класса

***internal*** Доступ ограничен текущей сборкой

***protectedinternal*** Доступ ограничен текущей сборкой или типами, которые являются производными от содержащего класса

***private*** Доступ ограничен содержащим типом

Инкапсуляция объединяет в классе описывающие этот объект поля данных и код, обрабатывающий эти данные. Для унификации использования классов родственных направлений принято работать с данными объектов через методы. Специфические методы, называемые *свойствами* объектов, позволяют не только изменять поля объекта, но и выполнять сопутствующие действия. Например, присвоение значения свойству графического объекта не только меняет соответствующие данные (поля объекта), но и меняет графическое изображение, такое как форма, размер, положение, цвет или контур. Единообразие именования свойств, методов и полей объектов схожих направлений, позволяющих использовать логически идентичные идентификаторы членов для разных реализаций, называется *полиморфизмом*.

***Полиморфизм***

**Полиморфизм** (греч., множество форм) – возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода. Общие свойства объектов объединяют 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*.

Примером полиморфизма может быть наличие у классов, описывающих графические объекты методов *Show(), Hide()*, *Move()* для соответственно отображения, скрытия и перемещения объекта. Объекты точки, прямоугольника и эллипса отрисовываются по-разному, но пользователю понятна запись *Ellipse1.Show()*, *Point1.Show()* и т.д. (показать эллипс, показать точку).

***Наследование***

**Наследование** – механизм ООП, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом.

В дочернем классе можно переписать функциональность метода, уже существующего в родительском классе, *скрывая* наследованные члены. Для этого строку описания перекрывающего метода необходимо начать с модификатора *new*. Для *расширения* реализации наследуемого члена вместо *скрытия* используется модификатор *override*. При наследовании нового класса от родительского, имя родительского класса указывается через двоеточие.

***Оператор new***

Перед обращением к объекту, его нужно создвть, т.к. класс – всего лишь абстракция, описание. Экземпляр объекта создается оператором *new*. Переменная экземпляра объекта является *ссылочной*, т.е. хранит ссылку на область памяти, в которой создан объект, а не сам объект. Присваивание переменной, ссылающейся на объект другой переменной такого же типа, заменит в ней ссылку на копию первой, что может быть причиной утериссылки на второй объект.

Поля и методы, имеющие модификатор *static*, создаются автоматически при старте программы и описывают статический член, принадлежащий всему типу, а не конкретному объекту. Модификатор *static* можно использовать с классами, полями, методами, свойствами, операторами, событиями и конструкторами, но нельзя – с индексаторами, деструкторами или типами, отличными от классов. К статическим методам относится метод *Main*(), который может быть только в одном классе, и с которого начинается программа.

***Конструкторы***

При создании объекта, обычно сразу заполняют его поля (присваивают значение), но при большом количестве полей некоторые из них можно пропустить. Такой прием, хотя и не запрещен, не применяют в профессиональном программировании. Для инициализации объектов применяют так называемые *конструкторы* – код, похожий на методы. У них отсутствует указанный явно возвращаемый тип, и их имя совпадает с именем класса. Кроме присвоения начальных значений полям объекта, конструкторы могут выполнять другие действия для формирования полноценного объекта. В классе могут присутствовать несколько конструкторов, различающихся количеством параметров. В случае отсутствия конструкторов, С# добавляет *конструктор по умолчанию*, который инициализирует все переменные объекта в значения по умолчанию (числа – в ноль, строки и другие ссылочные типы – в *null*, логические – *false*). После определения собственного конструктора, конструктор по умолчанию не используется. Также конструкторами по умолчанию называют явно заданные конструкторы, у которых отсутствуют параметры. У конструктора практически всегда указывается модификатор *public* для обеспечения доступа к конструктору извне класса.

Закрытие (*private*) конструкторы используется для явного обозначения невозможности создания экземпляра данного класса при отсутствии у класса полей или методов. Статический (*static*) конструктор используется для инициализации **статических** данных или выполнения определенного действия, которое необходимо выполнить только один раз, он вызывается автоматически (его нельзя вызвать явно) перед созданием первого экземпляра или ссылкой на какие-либо статические члены. Конструкторы не наследуются, т.к. при отсутствии конструкторов используется конструктор по умолчанию, в каждом классе необходимо явно описывать свои конструкторы.

**Задача 15.1** Рассмотрим пример, иллюстрирующий применение объектов (небольшие методы и конструкторы для экономии места записаны в одну строку).

*Текст программы:*

using System;

namespace C137

{

class Obj1

{

public double x, y;

public Obj1() : this(0.0, 0.0) { }

public Obj1(double X, double Y) { x = X; y = Y; }

public Obj1(Obj1 Copy) : this(Copy.x, Copy.y) { }

public double Sum() { return x + y; }

public double Op() { return Sum() + 1; }

}

class Obj2 : Obj1

{

public Obj2() : base() { }

public Obj2(double X, double Y) : base(X, Y) { }

public Obj2(Obj2 Copy) : base(Copy) { }

new public double Op() { return Sum() + 2; }

}

class Program

{

static void Main(string[] args)

{

Obj1 A1 = new Obj1(); A1.x = 1; A1.y = 2;

Obj2 A2 = new Obj2(3, 4); Obj1 A3;

Console.WriteLine("Obj1: " + A1.Sum() + " " + A1.Op() + "\nObj2: " + A2.Sum() + " " + A2.Op());

A3 = A1; Console.WriteLine("1: " + A3.x + " " + A3.y);

A3 = A2; Console.WriteLine("2: " + A3.x + " " + A3.y);

A3 = new Obj2(A2);

Console.WriteLine("3: " + A3.x + " " + A3.y);

Console.ReadLine();

}

}

}

Результаты расчета см. рис.15.1.



Рисунок 15.1

Пространство имен С137 содержит три класса: *Obj1*, *Obj2* и *Program*. Класс *Program* содержит один статический метод *Main*(), которому передается управление при запуске программы. Класс *Obj1* содержит два вещественных поля данных (*x* и *y*), доступных извне класса, т.к. они описаны модификатором *public*. Ниже описаны три *перегружаемых* конструктора с различным количеством параметров.

Первый, без параметров – *конструктор по умолчанию*, инициализирует все поля в нули, вызывая конструктор с двумя параметрами. Вызов производится с помощью служебного слова *this*(), обозначающего текущий класс, и соответственно конструктор класса. После вызова перегружаемого конструктора указываются пустые фигурные скобки, в которых может быть расположен код, выполняемый после кода вызванного конструктора (метода). Второй конструктор – с перечнем переменных в качестве параметров, значения которых присваиваются полям. Третий конструктор – с параметром такого же объекта, что позволяет копировать все значения указанного объекта для получения копии в текущем объекте, получаемые объекты идентичны в момент создания нового объекта, но могут различаться в процессе работы. Также содержаться общие (*public*) методы *Sum*() и *Op*(). Первый возвращает сумму *x* и *y*, второй – добавляет к сумме единицу.

Второй класс *Obj2* наследуется от класса *Obj1*, поэтому *Obj2* содержит те же поля и методы, что и *Obj1*. Т.к. конструкторы не наследуются, они описаны явно, но использование ключевого слова *base* позволяет обратиться к коду родительского объекта с соответствующими параметрами, поэтому сам код конструкторов отсутствует. Для демонстрации в классе *Obj2* наследуемый метод класса *Obj1 Op*() был скрыт модификатором *new* и переписан с новым кодом (вместо единицы добавляет двойку).

При запуске программы выполняется код статического метода *Main*(), в котором объявляется ссылочная переменная *А*1 типа *Obj1* и присваивается ей ссылка на созданный в памяти оператором *new* экземпляр объекта (по шаблону класса *Obj1*). При создании объекта используется конструктор по умолчанию, обнуляющий значения полей. Затем полям экземпляра объекта *А*1 присваиваются значения. Имена полей (как и любых других членов) соединяются с именами содержащих их объектов точкой. Такая запись присвоения полям значений допустима, но не приветствуется, т.к. менее понятна, громоздка и прямое обращение к полям может привести к ошибкам.

При создании экземпляра *А*2 класса *Obj2* используется конструктор, инициализирующий значения полей значениями аргументов, данный метод является предпочтительным.

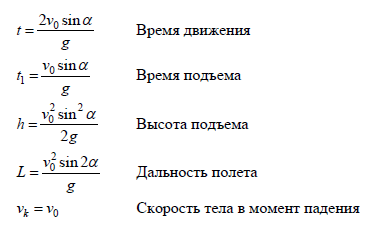
Ссылочная переменная *А*3 класса *Obj1* только описывается, но ей не присваивается никакого значения.

Далее оператором *WriteLine* выводятся результаты работы методов объектов (для второго объекта результат выводится с новой строки "\n"). Заметьте, что у *А*2 наследуемый от *Obj1* метод *Sum*() производит вычисления со «своими» данными, как и метод *Op*().

Далее демонстрируется присвоение ссылки на объекты. Переменной *А*3 присваивается ссылка на объект, хранящаяся в *А*1, и выводится значение полей первого объекта. При этом *А*1 и *А*3 ссылаются на один и тот же объект. Затем переменной *А*3 присваивается ссылка на второй объект, хранящаяся в *А*2 и соответственно выводятся значения полей второго объекта. Соответственно ссылки *А*2 и *А*3 ссылаются на один и тот же объект.

И в конце програмы переменной *А*3 присваивается ссылка на создаваемый новый экземпляр объекта класса *Obj2.*

**Задача 15.2** Тело бросили под углом *а* к горизонту со скоростью *v*0 . Найти время движения t, дальность полета L, максимальную высоту подъема h, скорость тела в момент падения *vk* . Класс, описывающий бросаемые объекты, инкапсулирует данные объекта и методы, их обрабатывающие. Конструктор класса должен инициализировать входные данные и вызвать методы, вычисляющие искомые данные.

**

*Текст программы:*

using System;

namespace C138

{

class Obj

{

public double v0, vk, L, t, t1, h, alfa;

public const double g = 9.8;

public Obj(double a, double v)

{

alfa = a \* Math.PI / 180; v0 = v;

vk = v0; t1 = Time1(); t = Time();

L = Length(); h = Height();

}

118

public double Length()

{

return v0 \* v0 \* Math.Sin(2 \* alfa) / g;

}

public double Height()

{

return v0 \* v0 \* Math.Pow(Math.Sin(alfa), 2.0) / 2.0 / g;

}

public double Time1()

{

return v0 \* Math.Sin(alfa) / g;

}

public double Time()

{

return Time1() \* 2;

}

}

class Program

{

static void Result(int n, Obj R)

{

String Format = "A{0} пролетел {1:#.##} м " +

"за {2:#.##} с, поднявшись на {3:#.##} м "+

"(v0={4:#.##}, a={5:#.##})";

Console.WriteLine(Format,n,R.L,R.t,R.h,R.v0,R.alfa);

}

static void Main(string[] args)

{

Obj A1=new Obj(45.0, 15.0);

Obj A2=new Obj(15.0, 45.0);

Result(1,A1);

Result(2,A2);

Console.ReadLine();

}

}

}

Результаты расчета см. рис.15.2.



Рисунок 15.2