**Курс лекций**

**«Объектно-ориентированное программирование»**

**1 Объектно-ориентированное программирования на языке C#**

Объектно-ориентированное (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс представляет собой шаблон, по которому определяется форма объекта. В нем указываются данные и код, который будет оперировать этими данными. В С# 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 во время исполнения программы.

Основные принципы объектно-ориентированного программирования:

Абстрагирование — выделение набора значимых характеристик объекта, исключая из рассмотрения незначимые.

Инкапсуляция — объединение данных и методов, работающих с ними, в классе с целью скрытия деталей реализации от пользователя класса (под пользователем класса понимается программист, который при написании программы использует возможности класса через методы класса (интерфейс) не зная деталей реализации).

Наследование — возможность создавать новые определения классов на основе существующих, расширяя и переопределяя их функциональность. Наследование используется для повторного использования кода. Класс, от которого производится наследование, называется базовым, родительским или суперклассом, новый класс — потомком, наследником или производным классом.

Полиморфизм — возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм позволяет поддерживать выполнение нужного действия в зависимости от типа объекта и применяется для универсальной обработки схожих объектов разных типов.

**Класс**

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными.

Синтаксис объявления класса, следующий:

**class** имя\_класса {

// Объявление полей.

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 методы (параметры) {

// тело метода

}

}

Модификатор доступа определяется одним из следующих ключевых слов:

private — (закрытый) поле или могут быть использован только внутри тела класса, в котором объявлены;

public — (открытый) поле или метод могут быть использован из любого места в программе.

Для создания объекта определенного класса используется оператор new.

Имя\_класса название\_объекта = new Имя\_класса();

Пример описания класса «Человек»

class Chelovek

{

public string surname;

public string name;

public int age;

}

Для использования класса Chelovek надо использовать класс Program, с метода Main которого начинается работа программы:

class Program

{

static void Main(string[] args)

{

Chelovek s = new Chelovek();

s.surname = "Иванов";

s.name = "Иван";

s.age = 20;

Console.WriteLine("Фамилия: "+s.surname);

Console.WriteLine("Имя: " + s.name);

Console.WriteLine("Возраст: " + s.age);

}

}

В методе Main создается объект s типа Chelovek и инициализируются поля.

После этого на экран выводится информация о всех полях объекта.

**Массив объектов**

Бывает необходимо поределить не один объект, а сразу несколько, поместив их в массив.

Chelovek[] group = new Chelovek[21];

group[0] = new Chelovek();

group[0].surname = "Иванов";

group[0].name = "Иван";

group[0].age = 20;

Console.WriteLine("Фамилия: " + group[0].surname);

Console.WriteLine("Имя: " + group[0].name);

Console.WriteLine("Возраст: " + group[0].age);

Chelovek c1 = new Chelovek();

Эта строка объявления выполняет три функции. Во-первых, объявляется переменная c1, относящаяся к классу Chelovek. Сама эта переменная не является объектом, а лишь переменной, которая может ссылаться на объект. Во-вторых, создается конкретное физическое представление объекта. Это делается с помощью оператора new. И наконец, переменной с1 присваивается ссылка на данный объект. Таким образом, после выполнения анализируемой строки объявленная переменная с1 ссылается на объект типа Chelovek.

Схема.

Пример, что будет если не написать new.

static void Main(string[] args)

{

Chelovek c1 = new Chelovek();

c1.Family = "Иванов";

c1.Name = "Иван";

Chelovek c2 = c1;

//Chelovek c2 = new Chelovek();

c1.Family = "Петров";

c1.Name = "Пётр";

c2.Family = "Сидоров";

c2.Name = "qwerty";

Console.WriteLine("c1: " + c1.Family + " " + c1.Name);

Console.WriteLine("c2: " + c2.Family + " " + c2.Name);

}

**Методы класса**

Методы представляют собой подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях они предоставляют доступ к этим данным. Методы класса описываются в классе и имеют доступ к полям этого класса. Как правило, другие части программы (другие классы) взаимодействуют с классом посредством его методов.

Метод состоит из одного или нескольких операторов. В грамотно написанном коде С# каждый метод выполняет только одну функцию (имеет одно назначение). У каждого метода имеется свое имя, по которому он вызывается. В общем случае, методу в качестве имени можно присвоить любой допустимый идентификатор. Следует, однако, иметь в виду, что идентификатор Main () зарезервирован для метода, с которого начинается выполнение программы. Кроме того, в качестве имен методов нельзя использовать ключевые слова С#.

Общая форма определения метода:

доступ тип имя(тип имя\_параметра, тип имя\_параметра, …)

{

// тело метода

}

тип — один из типов данных, имеющихся в C# (int, double, string и т.д.). Определяет тип данных результата выполнения действий в методе.

Помимо стандартных типов метод может иметь тип void (пустой). В этом случае он не возвращает никакого результата.

Список параметров — это последовательность пар, состоящих из типа и имени, разделенных запятыми. Параметры представляют собой переменные, получающие значение аргументов, передаваемых методу при его вызове.

Для возврата значения из метода в вызывающую часть программы служит следующая форма оператора return:

return значение;

где значение — это конкретное возвращаемое значение.

Если метод не возвращает никакого значения, то указывается тип void.

Пример с классом студент

class Program

{

static void Main(string[] args)

{

Student s = new Student();

Console.Write("Введите фамилию: ");

s.surname = Console.ReadLine();

Console.Write("Введите имя: ");

s.name = Console.ReadLine();

Console.Write("Введите год рождения: ");

s.year\_of\_birth = int.Parse(Console.ReadLine());

Console.WriteLine("Возраст = " + s.GetAge());

}

}

class Student

{

public string surname;

public string name;

public int year\_of\_birth;//было age, но возраст постоянно меняется

public int GetAge()

{

return DateTime.Now.Year - year\_of\_birth;

}

}

Пример — класс «Прямоугольник»

class Program

{

static void Main(string[] args)

{

Прямоугольник p = new Прямоугольник();

p.длина = 4;

p.ширина = 5;

Console.WriteLine("Периметр " + p.Периметр());

Console.WriteLine("Площадь " + p.Площадь());

}

}

class Прямоугольник

{

public double длина;

public double ширина;

public double Площадь()

{

return длина \* ширина;

}

public double Периметр()

{

return 2 \* длина + 2 \* ширина;

}

}

Пример — класс «Время»

class Program

{

static void Main(string[] args)

{

Time clock = new Time();

clock.SetTime(20, 58, 0);

Console.WriteLine(clock.GetTime());

for (int i = 0; i < 200; i++)

{

clock.Next();

Console.WriteLine(clock.GetTime());

}

}

}

class Time

{

private uint hour = 0;

private uint minute = 0;

private uint second = 0;

public void SetHour(uint h)

{

if (h < 24)

{ hour = h;

}

else

{

Console.WriteLine("Неверное значение часов");

}

}

public void SetMinute(uint m)

{

if (m < 60)

{

minute = m;

}

else

{

Console.WriteLine("Неверное значение минут");

}

}

public void SetSecond(uint s)

{

if (s < 60)

{

second = s;

}

else

{

Console.WriteLine("Неверное значение секунд");

}

}

public void SetTime(uint hour, uint minute, uint second)

{

SetHour(hour);

SetMinute(minute);

SetSecond(second);

}

public string GetTime()

{

return hour.ToString("D2") + ":" + minute.ToString("D2") + ":" + second.ToString("D2");

}

public void Next()

{

++second;

if (second > 59)

{

++minute;

if (minute > 59)

{

++hour;

hour %= 24;

}

minute = minute % 60;

}

second = second % 60;

}

}

**Передача параметров по ссылке**

При передаче по значению в метод передаётся копия значения переменной, указанной в качестве параметра. Переменная в той части программы, из которой был вызван метод, не меняется.

static void Main(string[] args)

{

int a = 1;

Change(a);

Console.WriteLine(a); //Здесь будет 1

}

static void Change(int a)

{

a += 5;

}

При передаче по ссылке в метод передаётся ссылка на область памяти, в которой хранится переменная. Любые операции с ней внутри метода меняют значение исходной переменной в том месте программы, откуда был вызван метод.

static void Main(string[] args)

{

int a = 1;

Change(ref a);

Console.WriteLine(a); //Здесь будет 6

}

static void Change(ref int a)

{

a += 5;

}

**Перегрузка методов**

В С# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса, при условии, что их параметры объявляются по-разному. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов. Перегрузка методов относится к одному из способов реализации полиморфизма в С#.

public void SetFIO(string sn, string n, string fn)

{

surname = sn;

name = n;

father\_name = fn;

}

public void SetFIO(string sn, string n)

{

surname = sn;

name = n;

}

public void SetFIO(string sn)

{

surname = sn;

}

**Конструктор**

Конструктор — метод, вызываемый при создании объекта для инициализации значений его полей. У конструктора такое же имя, как и у его класса. Тип значения у конструктора не указывается.

Конструктор по умолчанию

public Time()

{

hour = 0;

minute = 0;

second = 0;

}

Конструктор с параметрами

public Time(uint hour, uint minute, uint second)

{

SetHour(hour);

SetMinute(minute);

SetSecond(second);

}

Ключевое слово this указывает, что обращение идёт именно к полям данного класса.

Конструктор копирования. Принимает в качестве параметра объект того же класса.

public Time(Time t)

{

hour = t.hour;

minute = t.minute;

second = t.second;

}

**Деструктор**

В языке С# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой "сборки мусора". Такой метод называется деструктором и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Следует, однако, сразу же подчеркнуть, что деструкторы — весьма специфические средства, применяемые только в редких, особых случаях. И, как правило, они не нужны. Но здесь они рассматриваются вкратце ради полноты представления о возможностях языка С#.

~имя\_класса()

{

// код деструктора

}

**Перегрузка операторов**

Перегрузка операторов позволяет описывать и применять к созданным программистом типам данных (классам) операции, по смыслу эквивалентные уже имеющимся в языке.

Как только для класса определяются операторы, появляется возможность оперировать объектами этого класса, используя обычный синтаксис выражений в С#.

Общая форма перегрузки унарного оператора:

public static возвращаемый\_тип operator op(тип\_параметра операнд)

{

// операции

}

Общая форма перегрузки бинарного оператора:

public static возвращаемый\_тип operator op(тип\_параметра1 операнд1, тип\_параметра2 операнд2)

{

// операции

}

Возвращаемый\_тип может быть любым типом, но обычно он соответствует типу класса, для которого перегружается оператор.

Тип\_параметра для унарных операторов должен быть таким же, как и у класса, для которого перегружается оператор. А в бинарных операторах хотя бы один из операндов должен быть такого же типа, как и у его класса.

Модификатор static используется для объявления статического члена, принадлежащего собственно типу, а не конкретному объекту.

Доступные для перегрузки операторы

Бинарные

+ , –, \*, /, %, &, | , ^

Унарные

–, !, ++, ––

Операторы сравнения можно перегружать только парами.

== !=

> <

>= <=

Тип возвращаемого значения для операторов сравнения должен быть логическим (bool).

Добавить в пример со временем.

public static bool operator >(Time t1, Time t2)

{

if (t1.hour > t2.hour)

{

return true;

}

else if (t1.hour == t2.hour && t1.minute > t2.minute)

{

return true;

}

else if (t1.hour == t2.hour && t1.minute == t2.minute && t1.second > t2.second)

{

return true;

}

else

{

return false;

}

}

public static bool operator <(Time t1, Time t2)

{

if (t1.hour < t2.hour)

{

return true;

}

else if (t1.hour == t2.hour && t1.minute < t2.minute)

{

return true;

}

else if (t1.hour == t2.hour && t1.minute == t2.minute && t1.second < t2.second)

{

return true;

}

else

{

return false;

}

}

public static bool operator ==(Time t1, Time t2)

{

if (t1.hour == t2.hour && t1.minute == t2.minute && t1.second == t2.second)

{

return true;

}

return false;

}

public static bool operator !=(Time t1, Time t2)

{

if (t1.hour == t2.hour && t1.minute == t2.minute && t1.second == t2.second)

{

return false;

}

return true;

}

public static Time operator ++(Time t)

{

t.Next();

return t;

}

**Операторы преобразования**

В случае если в одном выражении необходимо использовать данные разных типов, с ними выполняются преобразования типов.

double a = 1;

int b = a; // неявное преобразование

Здесь требуется преобразование типа double в тип int, но так как при этом возможна потеря части данных (дробной части), то выполнить его просто так невозможно. Нужно указать это явно:

double a = 1;

int b = (int)a; // явное преобразование

Существуют две формы операторов преобразования: явная и неявная. Ниже они представлены в общем виде:

public static explicit operator целевой\_тип(исходный\_тип v) // явная

{

return значение;

}

public static implicit operator целевой\_тип(исходный\_тип v) // неявная

{

return значение;

}

Пример

public static implicit operator string(Time t)

{

return t.hour.ToString("D2") + ":" + t.minute.ToString("D2") + ":" + t.second.ToString("D2");

}

**Свойства**

Еще одной разновидностью члена класса является свойство. Как правило, свойство сочетает в себе поле с методами доступа к нему. Как было показано в приведенных ранее примерах программ, поле зачастую создается, чтобы стать доступным для пользователей объекта, но при этом желательно сохранить управление над операциями, разрешенными для этого поля, например, ограничить диапазон значений, присваиваемых данному полю. Этой цели можно, конечно, добиться и с помощью закрытой переменной, а также методов доступа к ее значению, но свойство предоставляет более совершенный и рациональный путь для достижения той же самой цели.

Свойство состоит из имени и аксессоров get и set. Аксессоры служат для получения и установки значения переменной. Главное преимущество свойства заключается в том, что его имя может быть использовано в выражениях и операторах присваивания аналогично имени обычной переменной, но в действительности при обращении к свойству по имени автоматически вызываются его аксессоры get и set.

тип имя{

get

{

// код аксессора для чтения из поля

}

set

{

// код аксессора для записи в поле

}

Для возврата значения из аксессора get используется оператор return, как и для методов.

Для получения значения, присваиваемое свойству используется ключевое слово value;

Пример со временем

private uint hour;

private uint minute;

private uint second;

public uint Hour

{

set

{

if (value < 24)

{

hour = value;

}

else

{

Console.WriteLine("Неверное значение часов");

}

}

get

{

return hour;

}

}

public uint Minute

{

set

{

if (value < 60)

{

minute = value;

}

else

{

Console.WriteLine("Неверное значение минут");

}

}

get

{

return minute;

}

}

public uint Second

{

set

{

if (value < 60)

{

minute = value;

}

else

{

Console.WriteLine("Неверное значение секунд");

}

}

get

{

return second;

}

}

Если удалить один из аксессоров, то соответствующее действие невозможно будет выполнить со свойством. Также для них можно задавать модификаторы доступа.

Пример с человеком

Только для чтения

class Program

{

static void Main(string[] args)

{

Chelovek c = new Chelovek();

c.familiya = "Иванов";

c.god\_rogdeniya = 1999;

Console.WriteLine(c.Vozrast);

}

}

class Chelovek

{

public string familiya;

public string imya;

public string otchestvo;

public int god\_rogdeniya;

public int Vozrast

{

get

{

return DateTime.Now.Year - god\_rogdeniya;

}

}

}

Для чтения и записи

class Program

{

static void Main(string[] args)

{

Chelovek c = new Chelovek();

c.familiya = "Иванов";

c.god\_rogdeniya = 1999;

Console.WriteLine(c.Vozrast);

c.Vozrast = 17;

Console.WriteLine(c.god\_rogdeniya);

}

}

class Chelovek

{

public string familiya;

public string imya;

public string otchestvo;

public int god\_rogdeniya;

public int Vozrast

{

get

{

return DateTime.Now.Year - god\_rogdeniya;

}

set

{

god\_rogdeniya = DateTime.Now.Year - value;

}

}

Пример с комплексными числами

class Program

{

static void Main(string[] args)

{

Complex c1 = new Complex(0, -4);

Complex c2 = new Complex(0, -4);

Complex c3 = c1 \* c1;

Console.WriteLine(c3);

}

}

class Complex

{

public double real; // реальная часть

public double imaginary; // мнимая часть

public Complex()

{

real = 0;

imaginary = 0;

}

public Complex(double r, double i)

{

real = r;

imaginary = i;

}

public Complex(double r)

{

real = r;

imaginary = 0;

}

public static Complex operator +(Complex c1, Complex c2)

{

Complex c3 = new Complex();

c3.real = c1.real + c2.real;

c3.imaginary = c1.imaginary + c2.imaginary;

return c3;

}

public static Complex operator -(Complex c1, Complex c2)

{

Complex c3 = new Complex();

c3.real = c1.real - c2.real;

c3.imaginary = c1.imaginary - c2.imaginary;

return c3;

}

public static Complex operator \*(Complex c1, Complex c2)

{

Complex c3 = new Complex();

c3.real = c1.real \* c2.real - c1.imaginary \* c2.imaginary;

c3.imaginary = c1.real \* c2.imaginary + c1.imaginary \* c2.real;

return c3;

}

public static Complex operator /(Complex c1, Complex c2)

{

Complex c3 = new Complex();

c3.real = (c1.real \* c2.real + c1.imaginary \* c2.imaginary) / (Math.Pow(c2.real, 2) + Math.Pow(c2.imaginary, 2));

c3.imaginary = (c2.real \* c1.imaginary - c2.imaginary \* c1.real) / (Math.Pow(c2.real, 2) + Math.Pow(c2.imaginary, 2));

return c3;

}

public static bool operator ==(Complex c1, Complex c2)

{

if (c1.real == c2.real & c1.imaginary == c2.imaginary) return true;

else return false;

}

public static bool operator !=(Complex c1, Complex c2)

{

if (c1.real != c2.real | c1.imaginary != c2.imaginary) return true;

else return false;

}

public static implicit operator string(Complex c)

{

return "(" + c.real + "," + c.imaginary + ")";

}

}

**Приложение**

**1. Объявление класса на языке C#**

Для реализации инкапсуляции в языке C# используются модификаторы доступа. Язык С# поддерживает следующие модификаторы доступа (модификаторы видимости):

—**public** — поля, свойства и методы являются общедоступными;

—**private** — поля, свойства и методы будут доступны только в классе, в котором они определены;

—**protected** — поля, свойства и методы будут доступны как в классе, в котором они определены, так и в любом производном классе;

— **internal** — поля, свойства и методы будут доступны во всех классах внутри сборки, в которой определен класс.

Для контроля доступа к полям класса в языке C# можно использовать свойства. Внутри класса свойство определяется в виде пары методов для присвоения значения свойств и для чтения значения свойства. Для пользователей объектов класса свойство представляется как поле класса. Синтаксис определения свойства класса:

<тип> <имя\_свойства> {

**get** {

**return** <значение>;

}

**set** {

<имя\_поля> = **value**;

}

}

Метод **get** используется для получения значения свойства. Метод **set** используется для изменения значения свойства. Внутри метода **set** новое значение свойства передается в виде переменной **value**. С помощью свойств можно проверить новое значение поля, прежде, чем изменить поле; выполнить дополнительные действия при изменении или чтении поля, представить для пользователя в виде поля информацию, которая на самом деле не является полем класса.

Язык C# не поддерживает множественное наследование.

Для реализации классического наследования в языке C# используется следующий синтаксис:

**class** <имя\_класса\_потомка> **:** <имя\_класса\_родителя> {

}

Для вызова в конструкторе класса-потомка конструктора класса-родителя используется ключевое слово **base**:

**class** <имя\_класса\_потомка> **:** <имя\_класса\_родителя> {

<имя\_класса\_потомка> (<аргументы>) **:** **base**(<аргументы>) {

}

}

Также ключевое слово **base** используется для вызовов методов класса-родителя из переопределенных методов в классе-потомке. Язык C# также поддерживает наследование включением.

Для реализации полиморфизма, основанного на виртуальных методах, в языке C# используются ключевые слова **virtual** и **override**:

**class** <имя\_класса\_родителя> {

**virtual** <имя\_метода1> (аргументы) {

}

}

**class** <имя\_класса\_потомка> **:** <имя\_класса\_родителя> {

**override** <имя\_метода1> (<аргументы>) {

}

}

Кроме того, язык C# поддерживает объявление абстрактных методов с помощью ключевого слова **abstract**. Абстрактные методы не содержат реализации и должны быть переопределены с помощью ключевого слова **override** в классах-потомках. Класс, в котором объявлен хотя бы один абстрактный метод, является абстрактным и должен быть объявлен с помощью ключевого слова **abstract**. Объекты абстрактного класса не могут быть созданы.

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными.

**class** <имя\_класса> {

// Объявление переменных.

<модификатор\_доступа> <имя\_типа> <имя\_переменной1>;

<модификатор\_доступа> <имя\_типа> <имя\_переменной2>;

//...

<модификатор\_доступа> <имя\_типа> <имя\_переменнойN>;

// Объявление методов.

<модификатор\_доступа> <возращаемый\_тип> <имя\_метода1> (<параметры>) {

<тело метода1>

}

<модификатор\_доступа> <возращаемый\_тип> <имя\_метода2> (<параметры>) {

<тело метода2>

}

//...

<модификатор\_доступа> <возращаемый\_тип> <имя\_методаN> (<параметры>) {

<тело методаN>

}

}

**2. Свойства класса**

Для удобства программирования в класс введено понятие свойства. Свойство представляет собой совокупность поля и одного или двух методов для считывания и/или записи данных в поле. Это позволяет использовать более предпочтительный синтаксис обращения к свойству как к полю, а не через синтаксис метода и одновременно позволяет скрыть данные в классе.

Так в языке Java свойств нет и обращение, например, к полю m\_*Fam* осуществляется через методы следующим образом:

**string** *s*;

stud *a* = **new** stud();

*s*=*a*.getFam();

*a*.setFam(s).

Метод getFam служит для считывания информации, записанной в поле m\_Fam, а метод setFam служит для записи нового значения в поле m\_Fam.

В данном случае класс имеет вид:

**class** stud

{

**private** **string** m\_Fam;

**public** **string** getFam (**void**) { **return** m\_Fam; }

**public** **void** setFam (**string** s);

}

В языке C# свойство позволяет использовать более удобный синтаксис обращения к свойству (как будто это поле):

**string** *s*;

stud *a*= **new** stud();

*s* = *a*.Fam;

*a*.Fam = *s*;

Синтаксис описания свойства следующий:

**class** stud

{

**private string** *m\_Fam*;

**public string** *Fam*

{

**get** { **return** *m\_Fam*; }

**set** { *m\_Fam*= **value**; }

}

}

Как видно полю *m\_Fam* сопоставлено свойство с двумя методами: метод **get** — для считывания данных из поля и метод **set** — для записи данных в поле. Свойство является удобным способом предотвратить некорректную работу с полем, например, запись информации в неверном формате или неверном диапазоне:

**public string** *Fam*

{

**get** { **return** *m\_Fam*; }

**set**

{

//поле реально изменится, если длина больше 1 буквы

//и первая буква русская строчная

**if** (**value**.length >1 & **value**[0]>=’А’ & **value**[0]<=’Я’)

*m\_Fam* = **value**;

}

}

Начиная с версии C# 3.0 имеется возможность использовать короткий синтаксис объявления свойства:

**public string** *Fam* { **get**; **set**; }

В таком синтаксисе нет возможности объявить тела методов **get** и **set**. Они принимаются по умолчанию

**get** { **return** *m\_Fam*; }

**set** { *m\_Fam* = **value**; }

Это не позволяет добавить проверку значения при записи в поле. Поле также описывать не нужно.

Например, класс Grajdanin содержит 3 свойства типа **string**:

**class** Grajdanin

{

**public string** *Fam* { **get**; **set**; } //фамилия

**public string** *Nam* { **get**; **set**; } //имя

**public string** *Country* { **get**; **set**; } //страна

}

Запись

**public string** *Fam* { **get**; **set**; } //фамилия

вводит свойство *Fam* и автоматически создаваемое поле и является более компактной и удобной формой ранее изучаемой записи

**3. Методы класса и методы объекта.**

Функциональность класса реализуется в методах. Методы могут быть статическими (методами класса) и методами объекта. Методы могут обрабатывать данные: поля класса и аргументы метода. Аргументы могут быть входными, аргументами ссылками (**ref)** и выходными аргументами (**out**).

Входные аргументы используются для передачи методу переменной по значению — в переменную метода передается копия аргумента. При передаче значения в функцию можно передать константное выражение.

**void** *func*(**int** *a*)

{

*a*++;

}

*func* (4); //можно так вызвать функцию *func*

**int** *c*=0;

*func* (*c*);

*Console.Write(c)*; //будет 0

Однако для объектов (экземпляров класса) передача по значению всегда означает передачу по ссылке (для экономии памяти):

**void** *func*(Grajdanin *a*)

{

*a.Fam =* ”Иванов”;

}

**Stud** *c*=new Grajdanin(“Козлов”);

**Console***.*Write*(c.Fam)*; //будет “Иванов”

Аргументы-ссылки (**ref)** позволяют передать значение по ссылке, которое метод мог бы изменить. Перед передачей аргумента **ref** в метод переменную надо инициализировать. Переменная должна иметь тот же тип, что и аргумент. Передавать константное выражение недопустимо.

**void** *func*(**ref int** *a*)

{

*a*++;

}

**int** *c*=0;

*func* (**ref** *c*); //будет *c*=1

*func* (**ref** 0); //нельзя — ошибка компиляции

Выходные аргументы (**out)** предназначены для возврата значения из метода. Отличие от параметра **ref** в том, что предварительная инициализация переменной не требуется.

**void** *Sq*(**float** *a*, **out float** *sq*)

{

*sq* = *a*\**a*;

}

**float** *s*; //не инициализировано

*Sq*(4, **out** *s*); //будет *s*=16

Поскольку функция имеет только один результат, то выходные параметры (**out**) могут использоваться для возврата из функции большего количества значений.

В C# допускается совместное использование одного и того же имени двумя или более методами одного и того же класса. В этом случае говорят, что методы перегружаются, а сам процесс называется перегрузкой методов.

**Перегрузка методов.** Перегрузка методов в C# означает, что в классе можно определить несколько методов с одним и тем же именем при условии, что эти методы получают разное число или типы параметров.

Перегрузка методов относится к одному из способов реализации полиморфизма в C#. В общем, для перегрузки метода достаточно объявить разные его варианты, а об остальном позаботится компилятор. Но при этом необходимо соблюсти следующее важное условие: тип или число аргументов у каждого метода должны быть разными. Совершенно недостаточно, чтобы два метода отличались только типами результата. Они должны отличаться типами или числом своих аргументов. Когда вызывается перегружаемый метод, то выполняется тот его вариант, параметры которого соответствуют (по типу и числу) передаваемым аргументам. Модификаторы параметров **ref** и **out** также учитываются, когда принимается решение о перегрузке метода, но отличие между ними не столь существенно.

**class** *UserInfo*

**{**

// Перегружаем метод ui

**public void** ui() //1

{

Console.WriteLine("Пустой метод\n");

}

**public void** ui(**string** Name) //2

{

**Console**.WriteLine("Имя пользователя: {0}",Name);

}

**public void** ui(**string** Name, **string** Family) //3

{

}

**public void** ui(**string** Name, **string** Family, **byte** Age) //4

{

}

**public void** ui(**string** Name, **string** Family, **ref** **byte** Age) //5

{

}

}

**UserInfo** *user1* = **new** *UserInfo*();

// Разные реализации вызова перегружаемого метода

*user1*.ui(); //вызывается метод 1

*user1*.ui("Александр"); //вызывается метод 2

*user1*.ui("Александр", "Ерохин", 26); //вызывается метод 4

**byte** *y =* 20;

*user1*.ui("Александр", "Ерохин", **ref** *y*); //вызывается метод 5

**4. Конструкторы.**

Конструктор — метод, вызываемый при создании объекта.

В классе автоматически создается конструктор по умолчанию. Это правило выполняется для любого класса, не имеющего явно описанного конструктора. В случае, если программист описал хотя бы один конструктор в классе, то конструктор по умолчанию автоматически не создается.

**Перегрузка конструкторов.** Как и методы, конструкторы также могут перегружаться. Это дает возможность конструировать объекты самыми разными способами.

Одна из самых распространенных причин для перегрузки конструкторов заключается в необходимости предоставить возможность одним объектам инициализировать другие разными способами.

Когда приходится работать с перегружаемыми конструкторами, то иногда очень полезно предоставить возможность одному конструктору вызывать другой. В C# это дается с помощью ключевого слова **this**. Ниже приведена общая форма такого вызова:

*<*имя\_конструктора*>* (<параметры1>) **:** **this**(<параметры2>)

{

<тело конструктора, которое может быть пустым>

}

В исходном конструкторе сначала выполняется перегружаемый конструктор, список параметров которого соответствует критерию <параметры2*>*, а затем все остальные операторы, если таковые имеются в исходном конструкторе. Ниже приведен соответствующий пример:

**class** *UserInfo*

{

**public string** *Name*, *Family*;

**public byte** *Age*;

// Используем ключевое слово **this** для

// создания "цепочки" конструкторов

**public** *UserInfo*() : **this**("None","None",0)

{

}

**public** *UserInfo*(*UserInfo* *obj*) : **this**(*obj*.*Name*, *obj*.*Family*, *obj*.*Age*)

{

}

**public** *UserInfo*(**string** *Name*, **string** *Family*, **byte** *Age*)

{

**this**.*Name* = *Name*;

**this**.*Family* = *Family*;

**this**.*Age* = *Age*;

}

}

Вызывать перегружаемый конструктор с помощью ключевого слова **this** полезно, в частности, потому, что он позволяет исключить ненужное дублирование кода.

**5. Создание списка**

Для создания списка используется класс **List**< > —обобщенный класс.

Для создания списка для хранения граждан необходимо записать

**List**<*Grajdanin*> *list* = **new** **List**<*Grajdanin*>( ),

где List<*Grajdanin* > — класс, в котором может храниться только список объектов класса *Grajdanin* или его потомков.

При таком описании создается пустой список, в который можно добавлять объекты класса *Grajdanin*, используя метод Add класса **List**, и конструктор по умолчанию класса *Grajdanin*.

Добавление нового объекта класса *Grajdanin* в список записывается следующим образом:

*list*.Add(**new** *Grajdanin*());

*list*[0].Fam = “Иванов”;

*list*[0].Nam = “Иван”;

*list*[0].Country = “Россия”;

или инициализировать поля до добавления в список так:

*Grajdanin* *g* = **new** *Grajdanin*();

*g*.Fam = “Иванов”;

*g*.Nam = “Иван”;

*g*.Country = “Россия”;

*list*.Add(*g*);

или инициализированные конструктором:

*Grajdanin* *g* = **new** *Grajdanin*(“Иванов”, “Иван”, “Россия”);

*list*.Add(*g*);

*list*.Add(“Петров”, “Петр”, “Россия”);

**6.** **Сокращенная инициализация**

В языке C#, начиная с версии 2.5 (Visual Studio 2008), имеется возможность добавлять объекты в список и инициализировать поля с помощью более компактной и удобной записи, которая выполняет то же, что и предыдущий программный код.

**List**<*Grajdanin*> *list* = **new** **List**<*Grajdanin*>( )

{

**new** *Grajdanin* { Fam = "Иванов", Nam = "Иван", Country ="Russia"},

**new** *Grajdanin* { Fam = "Петров", Nam = "Петр", Country ="Russia"},

**new** *Grajdanin* { Fam = "Сидоров", Nam = "Семен", Country ="Germany"},

**new** *Grajdanin* { Fam = "Кизин", Nam = "Киз", Country ="Израиль"}

};

Обратите внимание на то, что новая коллекция заполняется непосредственно внутри фигурных скобок без использования метода Add класса **List**.

Данный синтаксис инициализации будет работать даже при отсутствии конструктора с тремя параметрами для класса **Grajdanin,** поскольку в этом случае не вызывается конструктор.

Класс **Grajdanin** с кострукторами и методом ToString() будет иметь, например, следующий вид:

**class** Grajdanin

{

**public string** Fam { **get**; **set**; }

**public string** Nam { **get**; **set**; }

**public string** Country { **get**; **set**; }

**public override string** ToString()

{

**return** Fam + " " + Nam + " " + Country;

}

// конструктор по умолчанию (в классе может быть только один)

**public** Grajdanin()

{

Fam = "Иванов"; Nam = "Иван"; Country = "Russia";

}

// конструктор копирования (в классе может быть только один)

**public** Grajdanin(Grajdanin *g*)

{

Fam = *g*.Fam; Nam = *g*.Nam; Country = *g*.Country;

}

// конструкторы с параметрами

// конструкторов может быть сколько угодно,

// если различаются типы или количество аргументов

**public** Grajdanin(**string** fam, **string** nam)

{

Fam = fam; Nam = nam; Country = "Russia";

}

**public** Grajdanin(**string** fam, **string** nam, **string** contry)

{

Fam = fam; Nam = nam; Country = contry;

}

}

**7. Перекрытие и сокрытие методов**

**Перекрытие методов** позволяет в производном классе переопределить методы базового класса, если они определены как виртуальные (**virtual**). В производном классе можно создать метод с тем же именем, теми же аргументами и той же доступностью, дописав ключевое слово **override**.

Важной особенностью перекрытия является вызов метода производного класса при приведении производного класса к базовому классу.

**class** *Rod*

{

**public virtual int** *f*(**int** *a*) { }

}

**class** *Reb*: *Rod*

{

**public override int** *f* (**int** *a*) { }

}

*Reb* *r* = **new** *Reb*( );

*r*.*f*(1); //вызывается метод *f* ребенка

((*Rod*) *r*).*f* (1); //и здесь вызывается метод *f* ребенка,

//поскольку *r* в действительности объект класса *Reb.*

То есть решение о выборе вызываемого метода принимается во время работы программы, а не при компиляции. Это сказывается на быстродействии программы, но позволяет, например, одним программным кодом вызывать различные методы.

**Преимущество перекрытия методов при использовании списков.** В ссылку на родителя можно записать ссылку на родителя и на любого потомка (ребенка). То есть:

Rod r = **new** Rod();

Rod r1 = **new** Reb();

То есть в массив или список с элементами типа родителя можно поместить элементы — ссылки на ребенка.

**List***<Rod>list=***new****List***();*

**List**.Add(**new** Rod());

**List**.Add(**new** Rod());

**List**.Add(**new** Reb());

**List***.*Add(**new**Reb());

**for**(**int** *i = 0; i<list.*Count*; i++*)

*list*.Draw();

В этом случае два раза будет вызван метод Draw родителя и два раза — ребенка. При этом программный код в цикле не нужно модифицировать независимо от того, в какой последовательности будут находиться объекты в списке.

**Сокрытие методов.** Даже если в базовом классе метод не был объявлен виртуальным (**virtual**), в производном классе все равно можно объявить другой метод с такой же сигнатурой. Новый метод, однако, не перекроет метод базового класса, а скроет метод базового класса. При этом компилятор, решая, какой метод вызвать, всегда будет рассматривать тип данных, на который указывает переменная, как тип данных, заданный при ее объявлении. То есть решение о том, какой метод вызвать решает компилятор.

Для скрытия метода базового класса необходимо к его определению добавить ключевое слово **new**. Этот модификатор подскажет компилятору, что программист знает о факте сокрытия:

**public class** *Rod*

{

**public string** *GetFunnyString*()

{

**return** "Родитель!";

}

}

**public class** *Reb* : *Rod*

{

**public new string** *GetFunnyString*()

{

**return** "Ребенок!”;

}

}

.

*Rod* *Rod1*;

*Reb Reb1*;

*Rod1* = **new** *Rod*();

**Console**.WriteLine(*Rod1*.*GetFunnyString*()); // будет Родитель!

*Rod1* = **new** *Reb*();

**Console**.WriteLine(*Rod1*.*GetFunnyString*()); // будет Родитель!

*Reb2* = **new** *Reb*();

**Console**.WriteLine(*Reb2*.*GetFunnyString*()); // будет Ребенок!

}

**8. Абстрактные классы и абстрактные методы**

**Абстрактные классы** — это классы, экземпляр которых создать невозможно. Абстрактные классы определяются с модификатором **abstract**. Они **могут** содержать абстрактные методы.

**Абстрактные методы** объявляются с модификатором **abstract**, не содержат тела метода. Абстрактный метод может быть перекрыт в производных классах как абстрактный или неабстрактный метод. Если хотя бы один метод в классе абстрактный, то и класс должен быть абстрактным.

**public abstract class** *Rod*

{

**public abstract int** *MyAbstractMethod*(); // тело отсуствует

**public abstract int** *MyAbstractMethod1*(); // тело отсуствует

...

**}**

**public abstract class** *Reb* : *Rod*

{

**public override int** *MyAbstractMethod*()

{

**return** 0;

}

**public abstract int** *MyAbstractMethod1*(); // тело отсуствует

}

Класс Reb остался абстрактным, так как один метод является абстрактным.

**public class** *Reb2* : *Rod*

{

**public override int** *MyAbstractMethod*()

{

**return** 0;

}

**public override int** *MyAbstractMethod1*()

{

**return** 1;

}

}

**9. Запечатанные классы**

Запечатанные (**sealed**) классы и методы можно рассматривать как противоположность абстрактным классам и методам. Объявление класса или метода запечатанным означает, что произвести наследование и перекрытие невозможно.

**sealed class** *FinalClass* // наследовать не возможно

{

}

**public class** *myClass*

{

// перекрытие в дальнейшем невозможно

**public sealed override** FinalMethod()

}

То есть у класса FinalClass не может быть наследников, а у класса myClass может быть наследник, но в нем нельзя перекрыть метод FinalMethod()*.*

**10. Интерфейсы**

Интерфейс — это тип, определяющий набор методов и свойств без реализации, и используемый для определения классов со сходной функциональностью. Интерфейс определяется с помощью ключевого слова interface. Синтаксис определения интерфейса:

**interface** <имя\_интерфейса> {

<имя\_типа\_метода> <имя\_метода> (<аргументы>);

<имя\_типа\_свойства> <имя\_свойства> { **get**; **set**;}

}

Особенности использования интерфейсов:

— все методы интерфейса по определению являются открытыми, при этом запрещено использовать в определении методов модификаторы доступа;

— тип интерфейса можно использовать в объявлении параметров методов и переменных, но создавать объекты типа интерфейс нельзя.

— вместо изменения уже используемого интерфейса следует воспользоваться наследованием интерфейса;

— интерфейсы реализуются с помощью классов. Под реализацией классом интерфейса понимается написание в классе программного кода для каждого из объявленных в интерфейсе методов и свойств. Для реализации интерфейса необходимо:

— после имени класса, реализующего интерфейс, поставить двоеточие и написать имя интерфейса (если в классе необходимо реализовать несколько интерфейсов, следует разделить их имена запятыми);

— включить в класс все методы и свойства, определенные в интерфейсе;

— для каждого реализованного метода и свойства указать модификатор доступа public.

Возможность реализации одним классом нескольких интерфейсов заменяет отсутствие множественного наследования. Для получения доступа к интерфейсу объекта применяются следующие способы:

— явное приведение типа (с помощью операции **()** ) —

**(**<имя\_интерфейса>**)** <имя\_объекта>;

— с помощью ключевого слова **as** —

<имя объекта> **as** <имя\_интерфейса>;

с помощью ключевого слова **is** —

**if** (<имя\_объекта> **is** <имя\_интерфейса>).

Для реализации наследования интерфейсов в языке C# используется следующий синтаксис:

**interface** <имя\_интерфейса >: <имя\_интерфейса\_родителя> {

<тело интерфейса>

}

**11. Вызов базовых версий методов.**

Ключевое слово **base** явно указывает компилятору, что происходит обращение к методу базового класса.

**public class** *CustomerAccount*

{

**public virtual decimal** *CalculatePrice* (*CustomerAccount* *account*)

{

**return** 2000M;

}

}

**public class** *GoldAccount* **:** *CustomerAccount*

{

**public override decimal** *CalculatePrice*(*CustomerAccount* *account*)

{

**return base**.*CalculatePrice*(*account*) \* 0.9M;

}

}

**12. Исключения.**

В результате работы программы могут произойти непредвиденные ситуации, например, деление на 0 или переполнение разрядной сетки переменной.

Далеко не всегда ошибки случаются по вине того, кто кодирует приложение. Иногда приложение генерирует ошибку из-за действий конечного пользователя, или же ошибка вызвана контекстом среды, в которой выполняется код. В любом случае программист всегда должен ожидать возникновения ошибок в своих приложениях и проводить кодирование в соответствии с этими ожиданиями.

В .NET Framework предусмотрена развитая система обработки ошибок. Механизм обработки ошибок С# позволяет закодировать пользовательскую обработку для каждого типа ошибочных ситуаций, а также отделить код, потенциально порождающий ошибки, от кода, обрабатывающего их.

Основу обработки исключительных ситуаций в С# составляет пара ключевых слов **try** и **catch**. Эти ключевые слова действуют совместно и не могут быть использованы порознь. Ниже приведена общая форма определения блоков **try**/**catch** для обработки исключительных ситуаций:

**try** {

// Блок кода, проверяемый на наличие ошибок.

}

**catch** (<тип\_исключения> *exOb*) {

// Обработчик исключения типа ExcepType1.

}

**catch** (<тип\_исключения> *exOb*) {

// Обработчик исключения типа ExcepType2.

}

...

где <тип\_исключения> — это тип возникающей исключительной ситуации. Когда исключение генерируется оператором try, оно перехватывается составляющим ему пару оператором catch, который затем обрабатывает это исключение. В зависимости от типа исключения выполняется и соответствующий оператор catch. Так, если типы генерируемого исключения и того, что указывается в операторе catch, совпадают, то выполняется именно этот оператор, а все остальные пропускаются. Когда исключение перехватывается, переменная исключения exOb получает свое значение. На самом деле указывать переменную exOb необязательно. Так, ее необязательно указывать, если обработчику исключений не требуется доступ к объекту исключения, что бывает довольно часто. Для обработки исключения достаточно и его типа.

Для включения проверки переполнения в локальном месте программы можно использовать ключевое слово **checked**, а для отлавливания исключений **try … catch**.

**private** **void** *button1\_Click*(**object** *sender*, EventArgs *e*)

{

**int** *a* = 20;

**int** *s*=1;

**try**

{

**for** (**int** *i* = 1; *i* <= *a*; *i*++)

**checked** { *s* \*= *i*; }

}

**catch** (**System**.OverflowException)

{

//этот код будет выполнен при возникновения исключения

//System.OverflowException - переполнения

**MessageBox**.Show("Слишком большое число");

}

**catch** ( )

{

//этот код будет выполнен при возникновения любого

//другого исключения, не отловленного выше

MessageBox.Show("Какое-то исключение");

}

Text = *s*.ToString();

}

Внутри оператора **try** помещается код, в котором могут генерироваться исключения, а в блоке **catch** они обрабатываются.

Проверка двух исключнений

**private** **void** button1\_Click(object sender, EventArgs e)

{

**int** a = 2000000;

**int** s=2;

**try**

{

a = a / s;

checked { a = a \* 100000000; }

}

**catch** (**System**.OverflowException)

{

MessageBox.Show("Переполнение");

}

**catch** (**System**.DivideByZeroException)

{

**MessageBox**.Show("Деление на ноль");

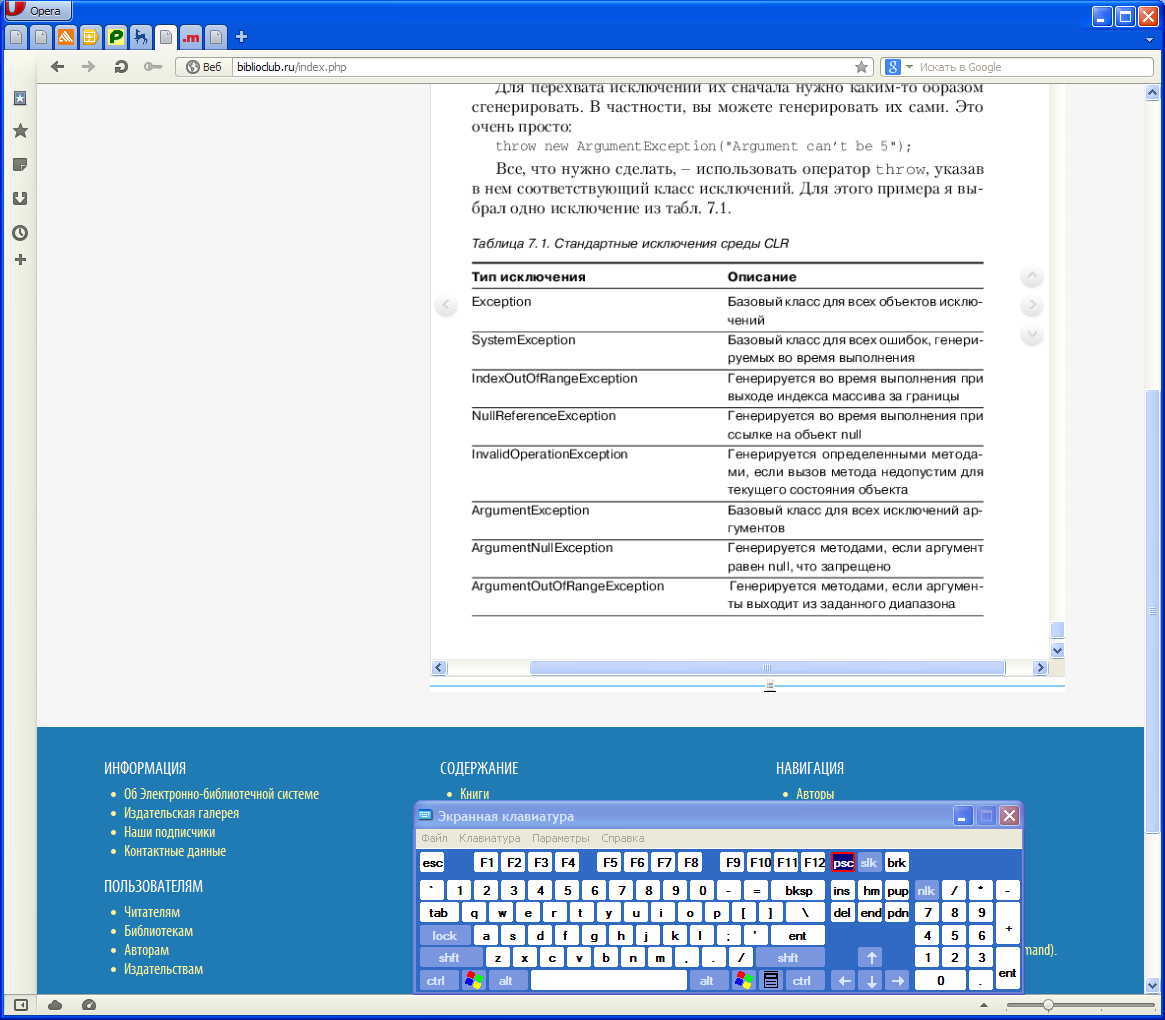
}

Text = a.ToString();

}

Если s=2, то произойдет исключение переполнения, если s=0 – то исключение деления на 0. Как только происходит исключение – прерывается выполнение блока try и управление передается в блок **catch**. То есть при делении на 0, умножение выполняться не будет и a останется равной 20.

Стандартные исключения



**Программная генерация исключений.**

Программист может прописать генерацию исключения с помощью ключевого слова **throw**, указав определенное исключение

**throw** **new** ArgumentOutOfRangeException(“Аргумент должен быть больше 5”);

**class** *Stol*{

**private float** *dl*;

**public float** *dlina*

{

**set**

{

**if** (**value**<=0)

**throw new**

ArgumentOutOfRangeException(“Аргумент должен быть больше 0”);

*dl*=**value**;

}

**get** {**return** *dl*; }

}

}

**class** Program

{

Stol s = **new** Stol( );

**void** main()

{

**try**{

s.dlina = -100;

}

**catch**(**ArgumentOutOfRangeException**)

{

MessageBox(“Длина не допустимая”);

s.dlina = 1;

}

}

}

Оператор [**throw**](https://msdn.microsoft.com/ru-ru/library/1ah5wsex.aspx) можно включить в блок **catch**, чтобы заново вызвать исключение, перехваченное блоком **catch**. В следующем примере извлекаются сведения об источнике из исключения [**IOException**](https://msdn.microsoft.com/ru-ru/library/system.io.ioexception.aspx), а затем это исключение вызывается для родительского метода:

**catch** (**FileNotFoundException** e)

{

// FileNotFoundExceptions are handled here.

}

**catch** (**IOException** e)

{

// Извлекаем информацию об исключении и вызываем родительский метод

// т.к. **IOException** родитель **FileNotFoundException**

**if** (e.Source != **null**)

**Console**.WriteLine("IOException source: {0}", e.Source);

**throw**;

}

Вы можете перехватывать одно исключение и сгенерировать другое исключение, как показано в следующем примере:

**catch** (**InvalidCastException** e)

{

// Выполняем некоторый код здесь и затем генерируем новое исключение

**throw new** YourCustomException ("Put your error message here.", e);

}

Также можно повторно вызывать исключение при выполнении указанного условия, как показано в следующем примере:

**catch** (**InvalidCastException** e)

{

**if** (e.Data = = **null**)

{

**throw**;

}

**else**

{

// Выполняем некоторые действия здесь

}

}

В блоке **try** инициализируйте только те переменные, которые в нем объявлены. В противном случае до завершения выполнения блока может возникнуть исключение. Например, в следующем примере кода переменная *n* инициализируется внутри блока **try**. Попытка использовать данную переменную вне этого блока **try** в инструкции Write(*n*) приведет к ошибке компилятора.

**static void** Main()

{

**int** *n*;

**try**

{

// Переменная *n* не описана в **try**, поэтому

// не надо инициализировать в **try …**

*n* = 123;

}

**catch**

{

}

// **…** так как в случае возникновения исключения здесь

//переменная *n* не будет инициализирована.

**Console**.Write(*n*);

**Вопросы для промежуточного контроля**

1. Понятие «класс».
2. Понятие «объект» или «экземпляр класса».
3. Основные понятия объектно-ориентированного программирования.
4. Понятие и назначение конструктора класса.
5. Элементы класса.
6. Модификаторы доступа к элементам класса.
7. Синтаксис описания полей класса.
8. Синтаксис описания конструкторов.
9. Синтаксис описания методов класса.
10. Понятие наследования и его описание.
11. Перегрузка методов.
12. Основные принципы объектно-ориентированного программирования.
13. Синтаксис описания свойств класса.
14. Понятие инкапсуляции.
15. Понятие наследования и его описание.
16. Перегрузка операторов.

**Список литературы**

1. Фленов М. Библия C#. – СПб.: БХВ-Петербург, 2011. – 560 с.
2. Вайсфейльд М. Объектно-ориентированный подход. – СПб.: Питер, 2020. – 256 с.
3. Зюзев А.М., Нестеров К.Е. Объектно-ориентированное программирование. – Екатеринбург: Из-во Уральского ун-та, 2019.
4. Гуськова О.И. Объектно-ориентированное программирование на Java. – М.: МПГУ, 2018. – 240 с.
5. Борков И.А. Объектно-ориентированное программирование. – СПб.: «Лань», 2019. – 700 с.