### Классы: основные понятия, данные, методы, конструкторы, свойства

#### Классы

##### Основные понятия

Класс – это структурированный тип данных объектного тина, объединяющий переменные и методы, работающие с ними и реализованный на принципах инкапсуляции, наследования и полиморфизма.

Класс — это обобщенное понятие, определяющие характеристики и поведение некоторого множества объектов, называемых экземплярами класса. "Классический" класс содержит данные, определяющие *свойства объектов класса*, и методы, определяющие их поведение. Для Windows-приложений в класс добавляется третья составляющая - события, на которые может *реагировать объект* класса. Все классы библиотеки .Net, а также все классы, которые создает программист в среде .Net, имеют одного общего предка – класс object.

Все программы, рассмотренные ранее, состояли из одного класса с одним методом Main и несколькими вспомогательными статическими методами. Теперь рассмотрим понятие "класс" более подробно.

Описание класса содержит ключевое слово class, за которым следует его *имя*, а далее в фигурных скобках – *тело* класса. Кроме того, для класса можно задать его базовые классы (предки) и ряд необязательных атрибутов и спецификаторов, определяющих различные характеристики класса:

[ атрибуты ] [ спецификаторы ] class имя\_класса [ : предки ]

{тело\_класса}

Простейший пример класса:

class Demo{}

Спецификаторы определяют свойства класса, а также доступность класса для других элементов программы. Возможные значения спецификаторов перечислены в следующей таблице:

|  |  |  |
| --- | --- | --- |
| **№** | **Спецификатор** | **Описание** |
| 1 | new | Задает новое описание класса взамен унаследованного от предка. Используется для вложения классов (в иерархии объектов). |
| 2 | public | Доступ к классу не ограничен |
| 3 | protected | Доступ только из данного или производного класса. Используется для вложенных классов. |
| 4 | internal | Доступ только из данной программы (сборки). |
| 5 | protected internal | Доступ только из данного и производного класса и из данной программы (сборки). |
| 6 | private | Доступ только из элементов класса, внутри которых описан данный класс. Используется для вложенных классов. |
| 7 | static | Статический класс. Позволяет обращаться к методам класса без создания экземпляра класса |
| 8 | sealed | «Бесплодный» класс. Запрещает наследование данного класса. Применяется в иерархии объектов. |
| 9 | abstract | Абстрактный класс. Применяется в иерархии объектов. |

Спецификаторы 2-6 называются спецификаторами *доступа*. Они определяют, откуда можно непосредственно обращаться к данному классу. Спецификаторы доступа могут комбинироваться с остальными спецификаторами.

**Замечание**. Атрибуты будут рассмотрены позже.

Класс можно описывать непосредственно внутри пространства имен или внутри другого класса. В последнем случае класс называется вложенным. В зависимости от места описания класса некоторые из этих спецификаторов могут быть запрещены. В данном разделе мы рассмотрим классы, которые описываются непосредственно в пространстве имен. Для таких классов допускаются только два спецификатора: public и internal. По умолчанию, то есть если ни один спецификатор доступа не указан, подразумевается спецификатор internal.

Объекты создаются явным или неявным образом, то есть либо программистом, либо системой. Программист создает экземпляр класса с помощью операции new, например:

Demo a = new Demo (); // Создается экземпляр класса Demo

Если достаточный для хранения объекта объем памяти выделить не удалось, то генерируется исключение OutOfMemoryException.

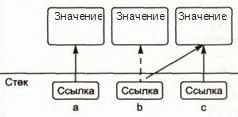
Для каждого объекта при его создании в памяти выделяется отдельная область, в которой хранятся его данные. В классе могут присутствовать *статические элементы*, которые существуют в единственном экземпляре для всех объектов класса. Статические данные часто называют данными класса, а остальные - данными экземпляра. Для работы с данными класса используются статические методы класса, для работы с данными экземпляра - *методы экземпляра*, или просто методы.

До сих пор мы использовали в программах только данные (переменные и константы) и методы. В общем случае класс может содержать следующие функциональные элементы:

1. Данные: переменные или константы.
2. Методы, реализующие не только вычисления, но и другие действия, выполняемые классом или его экземпляром.
3. Конструкторы (реализуют действия по *инициализации экземпляров* или класса в целом).
4. Свойства (определяют характеристики класса в соответствии со способами их задания и получения).
5. Деструкторы (определяют действия, которые необходимо выполнить до того, как объект будет уничтожен).
6. Индексаторы (обеспечивают возможность доступа к элементам класса по их порядковому номеру).
7. Операции (задают действия с объектами с помощью знаков операций).
8. События (определяют уведомления, которые может генерировать класс).
9. Типы (типы данных, внутренние по отношению к классу).

В данном разделе мы рассмотрим первые четыре категории элементов класса

Прежде чем приступить к проектированию классов, необходимо поговорить о присваивании и сравнении объектов. Механизм выполнения присваивания один и тот же для величин любого типа, как ссылочного, так и размерного, однако результаты различаются. При присваивании значения копируется значение, а при присваивании ссылки - ссылка, поэтому после присваивания одного объекта другому мы получим две ссылки, указывающие на одну и ту же область памяти:



Пусть были созданы три объекта а, b и с, а затем выполнено присваивание b = с. Теперь ссылки b и с указывают на один и тот же объект. Старое значение b становится недоступным и очищается сборщиком мусора.

Аналогичная ситуация с операцией проверки на равенство. Величины *значимого типа* равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные. Так, объекты b и с равны, т.к. они ссылаются на одну и ту же область памяти. Но, а не равно b даже при равенстве их значений.

##### ***Данные: поля и константы***

Данные, содержащиеся в классе, могут быть *переменными* или *константами* и задаются в соответствии с правилами, рассмотренными в теме "Идентификаторы". При описании данных также можно указывать атрибуты и спецификаторы, задающие различные характеристики элементов. Синтаксис описания элемента данных приведен ниже:

[атрибуты] [спецификаторы] [const] тип имя [ = начальное\_значение ];

Рассмотрим возможные спецификаторы для данных:

|  |  |  |
| --- | --- | --- |
| **№** | **Спецификатор** | **Описание** |
| 1 | new | Новое описание поля, скрывающее унаследованный элемент класса |
| 2 | public | Доступ к элементу не ограничен |
| 3 | protected | Доступ только из данного и производных классов |
| 4 | internal | Доступ только из данной сборки |
| 5 | protected internal | Доступ только из данного и производных классов и из данной сборки |
| 6 | private | Доступ только из данного класса |
| 7 | static | Одно поле для всех экземпляров класса |
| 8 | readonly | Поле доступно только для чтения (значение таких полей можно установить либо при описании, либо в конструкторе) |
| 9 | *volatile* | Поле может изменяться другим процессом или системой |

**Замечание**. Атрибуты будут рассмотрены позже.

Для констант можно использовать только спецификаторы 1-6.

По умолчанию элементы класса считаются закрытыми private. Для полей класса этот вид доступа является предпочтительным, поскольку поля определяют внутреннее строение класса, которое должно быть скрыто от пользователя. Все методы класса имеют непосредственный доступ к его закрытым полям.

Поля, описанные со спецификатором static, а также константы существуют в единственном экземпляре для всех объектов класса, поэтому к ним обращаются не через имя экземпляра, а через имя класса. Обращение к полю класса выполняется с помощью операции доступа (точка). Справа от точки задается имя поля, слева - имя экземпляра для обычных полей или имя класса для статических. Рассмотрим пример создания класса Demo и два способа обращения к его полям.

class Circle

{

public int x=0;

public int y=0;

public int radius=3;

public const double pi = 3.14;

public static string name = "Окружность";

double p;

double s;

}

class Program

{

static void Main()

{

Circle cr = new Circle(); //создание экземпляра класса

Console.WriteLine("pi=" + Circle.pi);// обращение к константе

Console.Write(Circle.name);// обращение к статическому полю

//обращение к обычным полям

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);

// Console.WriteLine(cr.p); - вызовет ошибку, т.к. поле p имеет тип private

Console.Write("Введите коэффициент= ");

int kof = int.Parse(Console.ReadLine());

cr.x -= kof; cr.y += kof; cr.radius \*= kof;

Console.WriteLine(" Новая окружность с центром в точке ({0},{1}) и радиусом {2}",

cr.x, cr.y, cr.radius);

//cr.s = 2 \* Circle.pi \* cr.radius; - вызовет ошибку, т.к. поле s имеет тип private

}

}

##### ***Методы***

**Замечание**. Создание и использование методов было рассмотрено нами ранее. Теперь рассмотрим использование методов в контексте создания классов.

Методы находятся в памяти в единственном экземпляре и используются всеми объектами одного класса совместно, поэтому необходимо обеспечить работу методов нестатических экземпляров с полями именно того объекта, для которого они были вызваны. Для этого в любой нестатический метод автоматически передается скрытый параметр this, в котором хранится ссылка на вызвавший функцию экземпляр.

В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызвавший объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода, например:

class Circle

{

public int x=0;

public int y=0;

public int radius=3;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle T() //метод возвращает ссылку на экземпляр класса

{

return this;

}

public void Set(int x, int y, int r)

{

this.x = x;

this.y = y;

radius=r;

}

}

class Program

{

static void Main()

{

Circle cr = new Circle(); //создание экземпляра класса

Console.WriteLine("pi=" + Circle.pi);// обращение к константе

Console.Write(Circle.name);// обращение к статическому полю

//обращение к обычным полям

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", cr.x, cr.y, cr.radius);

cr.Set(1, 1, 10);

Console.WriteLine("Новая окружность с центром в точке ({0},{1}) и радиусом {2}",

cr.x, cr.y, cr.radius);

Circle b=cr.T();//получаем ссылку на объект cr, аналог b=c

Console.WriteLine("Новая ссылка на окружность с центром в точке ({0},{1})

и радиусом {2}", b.x, b.y, b.radius);

}

}

##### ***Конструкторы***

Конструктор предназначен для инициализации объекта. Конструкторы делятся на конструкторы класса (для статических классов) и конструкторы экземпляра класса (всех остальных классов).

###### ***Конструкторы экземпляра***

Конструктор экземпляра вызывается автоматически при создании объекта класса с помощью операции new. Имя конструктора совпадает с именем класса. Рассмотрим основные свойства конструкторов:

1. Конструктор не возвращает значение, даже типа void.
2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации.
3. Если программист не указал ни одного конструктора или какие-то поля не были инициализированы, полям *значимых типов* присваивается нуль, полям ссылочных типов - значение null.

До сих пор мы задавали начальные значения полей класса при описании класса. Это удобно в том случае, когда для всех экземпляров класса начальные значения некоторого поля одинаковы. Если же при создании объектов требуется присваивать полю разные значения, это следует делать с помощью явного задания конструктора. В следующем примере добавлен конструктор и метод Print для вывода информации об объекте:

class Circle

{

public int x;

public int y;

public int radius;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle(int x, int y, int r)//конструктор

{

this.x = x;

this.y = y;

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора

a.Print();

Circle b=new Circle(10, 10, 5);//вызов конструктора

b.Print();

}

}

Часто бывает удобно задать в классе несколько конструкторов, чтобы обеспечить возможность инициализации объектов разными способами. Для этого конструкторы должны иметь разные сигнатуры. Рассмотрим это на примере:

class Circle

{

public int x;

public int y;

public int radius;

public const double pi = 3.14;

public static string name = "Окружность";

public Circle(int x, int y, int r)//конструктор 1

{

this.x = x;

this.y = y;

radius = r;

}

public Circle(int r)//конструктор 2

{

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора 1

a.Print();

Circle b=new Circle(5);//вызов конструктора 2

b.Print();

}

}

Обратите внимание на то, что в конструкторе 2 не были инициализированы поля x, y, поэтому им присваивается значение 0.

Если один из конструкторов выполняет какие-либо действия, а другой должен делать то же самое плюс еще что-нибудь, то удобно вызвать первый конструктор из второго. Для этого используется уже известное вам ключевое слово this в другом контексте, например:

public Circle(int x, int y, int r):this(r) //конструктор 1

{

this.x = x;

this.y = y;

}

public Circle(int r) //конструктор 2

{

radius = r;

}

Конструкция, находящаяся после двоеточия, называется *инициализатором*, то есть тем кодом, который исполняется до начала выполнения тела конструктора. В нашем случае конструктор 1 до выполнения своего кода вызывает конструктор 2.

###### ***Конструкторы класса***

Статические классы содержат только статические члены, в том числе и конструктор, которые хранятся в памяти в единственном экземпляре. Поэтому создавать экземпляры класса нет смысла.

Чтобы подчеркнуть этот факт, в первой версии С# для статических классов создавали два конструктора, один - пустой закрытый (private) конструктор, второй - статический конструктор, не имеющий параметров. Первый конструктор предотвращал попытки создания экземпляров класса. Второй конструктор автоматически вызывается системой до первого обращения к любому элементу статического класса, выполняя необходимые действия по инициализации. Вышесказанное отражено в следующем примере:

class Demo

{

static int a;

static int b;

private Demo(){} //закрытый конструктор

static Demo() //статический конструктор

{

a=10;

b=2;

}

public static void Print ()

{

Console.WriteLine("{0}+{1}={2}",a,b,a+b);

Console.WriteLine("{0}\*{1}={2}",a,b,a\*b);

Console.WriteLine("{0}-{1}={2}",a,b,a-b);

}

}

class Program

{

static void Main()

{

//Demo S=new Demo(); //ошибка содать экземпляр класса нельзя

Demo.Print();

}

}

В версию 2.0 введена возможность описывать статический класс, то есть класс с модификатором static. Экземпляры такого класса создавать запрещено, и кроме того, от него запрещено наследовать. Все элементы такого класса должны явным образом объявляться с модификатором static (константы и *вложенные типы* классифицируются как *статические элементы* автоматически). Конструктор экземпляра для статического класса задавать запрещается.

static class Demo

{

static int a=20;

static int b=10;

public static void Print ()

{

Console.WriteLine("{0}+{1}={2}",a,b,a+b);

Console.WriteLine("{0}\*{1}={2}",a,b,a\*b);

Console.WriteLine("{0}-{1}={2}",a,b,a-b);

}

}

class Program

{

static void Main()

{

Demo.Print();

}

}

##### ***Свойства***

Иногда требуется создать поле, которое с одной стороны, должно быть доступно для использования, с другой стороны, возможность что-то сделать с этим полем имеет ограничения. Например, полю нельзя присваивать произвольное значение, а только значения из какого-то диапазона. Свойство предлагает простой и удобный способ решения этой проблемы.

Синтаксис свойства:

[атрибуты] [спецификаторы] тип имя\_свойства

{

[get код\_доступа]

[set код\_доступа]

}

Значения спецификаторов для свойств и методов аналогичны. Чаще всего свойства объявляются как открытые (со спецификатором public ).

Код доступа представляет собой блоки операторов, которые выполняются при получении ( get ) или установке ( set ) свойства. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, то свойство доступно только для чтения. Если отсутствует часть get, то свойство доступно только для записи.

Рассмотрим пример работы со свойствами, обращая внимание на то, что синтаксически чтение и запись свойства выглядят почти как методы. При этом get должен содержать оператор return, возвращающий выражение, для типа которого должно существовать *неявное преобразование* к типу свойства.

class Circle

{

//закрытые поля

int x;

int y;

int radius;

public static string name = "Окружность";

public Circle(int x, int y, int r):this(r)//конструктор 1

{

this.x = x;

this.y = y;

}

public Circle(int r)//конструктор 2

{

radius = r;

}

public void Print()

{

Console.Write(name);

Console.WriteLine(" с центром в точке ({0},{1}) и радиусом {2}", x, y, radius);

Console.WriteLine();

}

public int X //свойство для обращения к полю x

{

get

{

return x;

}

set

{

x = value;

}

}

public int Y //свойство для обращения к полю y

{

get

{

return y;

}

set

{

y = value;

}

}

public int R //свойство для обращения к полю radius

{

get

{

return radius;

}

set

{

radius = value;

}

}

public double P //свойство только для чтения

{

get

{

return 2\* Math.PI \*radius;

}

}

public double S //свойство только для чтения

{

get

{

return Math.PI \*radius\*radius;

}

}

}

class Program

{

static void Main()

{

Circle a = new Circle(0, 0, 1); //вызов конструктора

a.Print();

//установка новых значений

a.X=1;

a.Y=1;

a.R=10;

//a.S=100; //ошибка - свойство доступно только для чтения

Console.WriteLine("центр=({0},{1})

радиус={2}

периметр={3:f2}

площадь={4:f2}",

a.X, a.Y, a.R, a.P, a.S);

}

}

### Классы: деструкторы, индексаторы, операции класса, операции преобразования типов

#### **Деструкторы**

В С# существует специальный вид метода, называемый деструктором, который вызывается сборщиком мусора непосредственно перед удалением объекта из памяти.

**Замечание**. Напоминаем, что сборщик мусора удаляет объекты, на которые нет ссылок. Он работает в соответствии со своей внутренней стратегией в неизвестные для программиста моменты времени.

В деструкторе описываются действия, гарантирующие корректность последующего удаления объекта. Например, проверяется все ли ресурсы, используемые объектом, освобождены (файлы закрыты, *удаленное соединение* разорвано и т. п.).

Синтаксис деструктора:

[атрибуты] [extern] ~имя\_класса()

{тело\_деструктора}

Деструктор не имеет параметров, не возвращает значения и не требует указания спецификаторов доступа. Его имя совпадает с именем класса и предваряется *тильдой* ( ~ ), символизирующей обратные по отношению к конструктору действия. Тело деструктора представляет собой блок или просто точку с запятой. Если деструктор определен как внешний, то используется спецификатор extern. Пример работы деструктора:

class DemoArray

{

int[] MyArray;//закрытый массив

string name; //закрытое поле

public DemoArray(int size,int x, string name)//конструктор

{

MyArray = new int[size];

this.name = name;

for (int i=0;i<size; ++i) MyArray[i]=x;

}

public void Print ()//метод

{

Console.Write(name+ " : ");

foreach (int a in MyArray) Console.Write(a+" ");

Console.WriteLine();

}

public int LengthN //свойство

{

get { return MyArray.Length; }

}

~DemoArray()//деструктор

{

Console.WriteLine("сработал деструктор для объекта "+this.name);

}

}

class Program

{

static void Main()

{

DemoArray a= new DemoArray(5,2, "один");

a.Print();

DemoArray b = new DemoArray(6,1, "два");

b.Print();

a = b;

a.Print();

}

}

**Замечание**. Обратите внимание на то, что деструкторы были вызваны автоматически.

В общем случае применение деструкторов замедляет процесс сборки мусора. Поэтому создавать деструкторы следует только тогда, когда необходимо освободить какие-то ресурсы перед удалением объекта.

Прежде чем рассматривать индикаторы, рассмотрим обработку исключений.

# **Обработка исключений**

Язык С#, как и многие другие объектно-ориентированные языки, реагирует на ошибки и ненормальные ситуации с помощью механизма обработки *исключений*. *Исключение* - это *объект*, генерирующий информацию о "необычном программном происшествии". При этом важно проводить различие между ошибкой в программе, ошибочной ситуацией и исключительной ситуацией.

*Ошибка в программе* допускается программистом при ее разработке. Например, вместо *операции* сравнения ( == ) используется *операция присваивания* ( = ). Программист должен исправить подобные ошибки до передачи кода программы заказчику. Использование механизма обработки исключений не является защитой от ошибок в программе.

*Ошибочная ситуация* вызвана действиями пользователя. Например, *пользователь* вместо числа ввел строку. Такая ошибка способна вызывать *исключение*. Программист должен предвидеть ошибочные ситуации и предотвращать их с помощью операторов, проверяющих допустимость поступающих данных.

Даже если программист исправил все свои ошибки в программе, предвидел все ошибочные ситуации, он все равно может столкнуться с непредсказуемыми и неотвратимыми проблемами - *исключительными ситуациями*. Например, нехваткой доступной памяти или попыткой открыть несуществующий *файл*. Исключительные ситуации программист предвидеть не может, но он может отреагировать на них так, что они не приведут к краху программы.

Для обработки ошибочных и исключительных ситуаций в С# используется специальная подсистема обработки исключений. Преимущество данной подсистемы состоит в автоматизации создания большей части кода по обработке исключений. Раньше этот код приходилось вводить в программу "вручную". Кроме этого, обработчик исключений способен распознавать и выдавать информацию о таких стандартных исключениях, как *деление* на нуль или попадание вне диапазона определения индекса.

#### **Оператор try**

В С# исключения представляются классами. Все *классы исключений* порождены от *встроенного класса* исключений Exception, который определен в пространстве имен System.

Управление обработкой исключений основывается на использовании оператора try. Синтаксис оператора:

try // контролируемый блок

{

…

}

catch //один или несколько блоков обработки исключений

{

…

}

finally //блок завершения

{

…

}

Программные инструкции, которые нужно проконтролировать на предмет исключений, помещаются в блок try. Если исключение возникает в этом блоке, оно дает знать о себе *выбросом* определенного рода информации. *Выброшенная информация* может быть перехвачена и обработана соответствующим образом с помощью блока catch. Любой код, который должен быть обязательно выполнен при выходе из блока try, помещается в блок finally. Рассмотрим пример, демонстрирующий, как отследить и перехватить исключение.

static void Main()

{

int x = int.Parse(Console.ReadLine());

int y =1 / x;

Console.WriteLine(y);

}

Перечислим, какие исключительные ситуации могут возникнуть:

1. пользователь может ввести нечисловое значение
2. если ввести значение 0, то произойдет деление на 0.

Теперь попробуем обработать эти ситуации. Для этого изменим код следующим образом.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

int y =1 / x;

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

Console.WriteLine("блок try выполнился успешно");

}

catch

{

Console.WriteLine("возникла какая-то ошибка");

}

Console.WriteLine("конец программы");

}

Рассмотрим, как обрабатываются исключения в данном примере. Когда возникает исключение, выполнение программы останавливается и управление передается блоку catch. Этот блок *никогда* не возвращает управление в то место программы, где возникло исключение. Поэтому команды из блока try, расположенные ниже строки, в которой возникло исключение, никогда не будут выполнены. Блок catch обрабатывает исключение, и выполнение программы продолжается с оператора, следующего за этим блоком.

В нашем случае при вводе нечислового значения или 0 будет выведено сообщение "возникла ошибка", а затем сообщение "конец программы".

Обработчик исключений позволяет не только отловить ошибку, но и вывести полную информацию о ней. Для демонстрации сказанного заменим блок catch следующим фрагментом.

catch (Exception error)

{

Console.WriteLine("Возникла ошибка {0}", error);

}

Теперь, если возникнет исключительная ситуация, "выброшенная" информация будет записана в идентификатор error. Данную информацию можно просмотреть с помощью метода WriteLine. Такое сообщение очень полное и будет полезно только разработчику на этапе отладки проекта.

Для пользователя на этапе эксплуатации приложения достаточно более краткой информации о типе ошибке. С этой целью в С# выделены стандартные *классы исключений*, такие как DivideByZeroException, FormatException. Внесем изменения в программу.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine()); // 1 ситуация

int y =1 / x; // 2 ситуация

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

Console.WriteLine("блок try выполнился успешно");

}

catch(FormatException) // обработка 1 ситуации

{

Console.WriteLine("Ошибка: введено нечисловое значение!");

}

catch (DivideByZeroException) // обработка 2 ситуации

{

Console.WriteLine("Ошибка: деление на 0!");

}

Console.WriteLine("конец программы");

}

В данном примере обрабатывается каждая ситуация в отдельности, при этом пользователю сообщается лишь минимальная информация об ошибке. В следующей таблице содержится описание наиболее часто используемых обработчиков стандартных исключений.

|  |  |
| --- | --- |
| **Имя** | **Описание** |
| ArithmeticException | Ошибка в арифметических операциях или преобразованиях |
| ArrayTypeMismatchException | Попытка сохранения в массиве элемента несовместимого типа |
| DivideByZeroException | Попытка деления на ноль |
| FormatException | Попытка передать в метод аргумент неверного формата |
| IndexOutOfRangeException | Индекс массива выходит за границу диапазона |
| InvalidCastException | Ошибка преобразования типа |
| OutOfMemoryException | Недостаточно памяти для нового объекта |
| OverflowException | Переполнение при выполнении арифметических операций |
| StackOverflowException | Переполнение стека |

Одно из основных достоинств обработки исключений состоит в том, что она позволяет программе отреагировать на ошибку и продолжить выполнение. Рассмотрим программу, которая строит таблицу значений для функции вида y(x)=100/(x2-1).

static void Main()

{

Console.WriteLine("a=");

int a = int.Parse( Console.ReadLine());

Console.WriteLine("b=");

int b = int.Parse(Console.ReadLine());

for (int i = a; i <= b; ++i)

{

try

{

Console.WriteLine("y({0})={1}", i, 100 / (i \* i - 1));

}

catch (DivideByZeroException)

{

Console.WriteLine("y({0})=Деление на 0", i);

}

}

}

Если встречается деление на нуль, генерируется исключение типа DivideByZeroException. В программе это исключение обрабатывается выдачей сообщения об ошибке, после чего выполнение программы продолжается. При этом попытка разделить на нуль не вызывает внезапную динамическую ошибку (т.к. блок обработки прерываний помещен внутрь цикла for ). Вместо этого исключение позволяет красиво выйти из ошибочной ситуации и продолжить выполнение программы.

#### **Операторы checked и unchecked**

В С# предусмотрено специальное средство, которое связано с генерированием исключений, вызванных переполнением результата в арифметических вычислениях. Например, когда значение арифметического выражения выходит за пределы диапазона, определенного для типа данных выражения. Рассмотрим небольшой фрагмент программы:

static void Main()

{

byte x = 200; byte y = 200;

byte result = (byte) (x + y);

Console.WriteLine(result);

}

Здесь сумма значений а и b превышает диапазон представления значений типа byte. Следовательно, результат данного выражения не может быть записан в переменную result, тип которой byte.

Для управления подобными исключениями в С# используются операторы checked и *unchecked*. Чтобы указать, что некоторое выражение должно быть проконтролировано на предмет переполнения, используйте ключевое слово checked. А чтобы проигнорировать переполнение, используйте ключевое слово *unchecked*. В последнем случае результат будет усечен так, чтобы его тип соответствовал типу-результату выражения.

**Замечание**. По умолчанию проверка переполнения отключена (галочка не стоит). В результате код выполняется быстро, но тогда программист должен быть уверен, что переполнения не случится или предусмотреть его возникновение. Как мы уже упоминали, можно включить проверку переполнения для всего проекта, однако она не всегда нужна. С помощью использования операторов checked и *unchecked* в С# реализуется механизм гибкого управления проверкой

Можно задать (или отключить) проверку переполнения сразу для всего проекта. Для этого необходимо выполнить следующие действия

1. Щелкнуть правой кнопкой мыши на имени проекта
2. В выпадающем меню выбрать Properties
3. В появившемся окне выбрать слева страницу Build
4. Щелкнуть на кнопке Advanced
5. В появившемся окошке поставить или убрать галочку напротив Check for *arithmetic overflow*/*underflow* property.

Оператор checked имеет две формы:

1. проверяет конкретное выражение и называется *операторной checked-формой*

checked ((тип-выражения) expr)

где expr — выражение, значение которого необходимо контролировать. Если значение контролируемого выражения переполнилось, генерируется исключение типа OverflowException.

1. проверяет блок инструкций
2. checked
3. {
4. // Инструкции, подлежащие проверке.

}

Оператор *unchecked* также имеет две формы:

1. операторная форма, которая позволяет игнорировать переполнение для заданного выражения

unchecked ((тип-выражения) expr)

где ехрr — выражение, которое не проверяется на предмет переполнения. В случае переполнения это выражение усекается.

1. игнорирует переполнение, которое возможно в блоке инструкций
2. unchecked
3. {
4. // Инструкции, для которых переполнение игнорируется.

}

Рассмотрим пример программы, которая демонстрирует использование checked и *unchecked*.

static void Main()

{

byte x = 200; byte y = 200;

try

{

byte result = unchecked((byte)(x + y));

Console.WriteLine("1: {0}", result);

result = checked((byte)(x + y));

Console.WriteLine("2: ", result);

}

catch (OverflowException)

{

Console.WriteLine("возникло переполнение");

}

}

*Результат выполнения программы*:

1: 144

возникло переполнение

В данном примере мы посмотрели, как использовать checked и uncheсked для проверки выражения. А теперь посмотрим, как использовать их для контроля за блоком инструкций.

static void Main()

{

byte n = 1; byte i;

try

{

unchecked //блок без проверки

{

for (i = 1; i < 10; i++) n \*= i;

Console.WriteLine("1: {0}", n);

}

checked //блок с проверкой

{

n=1;

for (i = 1; i < 10; i++) n \*= i;

Console.WriteLine("2: ", n);

}

}

catch (OverflowException)

{

Console.WriteLine("возникло переполнение");

}

}

Результат выполнения программы:

1: 128

возникло переполнение

##### ***Генерация собственных исключений***

До сих пор мы рассматривали исключения, которые генерирует среда, но сгенерировать исключение может и сам программист. Для этого необходимо воспользоваться оператором throw, указав параметры, определяющие вид исключения. Параметром должен быть объект, порожденный от стандартного класса System.Exception. Этот объект используется для передачи информации об исключении обработчику.

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

if (x < 0) throw new Exception(); //1

Console.WriteLine("ok");

}

catch

{

Console.WriteLine("введено недопустимое значение");

}

}

В строчке 1 c помощью команды new был создан объект исключения типа Exception. При необходимости можно генерировать исключение любого типа.

При генерации исключения можно определить сообщение, которое будет "выбрасываться" обработчиком исключений. Например:

static void Main()

{

try

{

int x = int.Parse(Console.ReadLine());

if (x < 0) throw new Exception("введено недопустимое значение"); //1

Console.WriteLine("ok");

}

catch (Exception error)

{

Console.WriteLine(error.Message);

}

}

#### **Индексаторы**

Индексатор представляет собой разновидность свойства и обычно применяется для организации доступа к скрытым полям класса по индексу, например, так же, как мы обращаемся к элементу массива. Синтаксис индексатора аналогичен синтаксису свойства:

[атрибуты] [спецификаторы] тип this [список параметров] /\* последние [ ] являются элементами синтаксиса\*/

{

[get код\_доступа]

[set код\_доступа]

}

Спецификаторы аналогичны спецификаторам свойств и методов. Индексаторы чаще всего объявляются со спецификатором public, поскольку они входят в *интерфейс объекта*. Атрибуты и спецификаторы могут отсутствовать.

Код доступа представляет собой блоки операторов, которые выполняются при получении ( get ) или установке ( set ) значения некоторого элемента класса. Может отсутствовать либо часть get, либо set, но не обе одновременно. Если отсутствует часть set, индексатор доступен только для чтения, если отсутствует часть get, индексатор доступен только для записи.

Список параметров содержит одно или несколько описаний индексов, по которым выполняется доступ к элементу. Чаще всего используется один индекс целого типа.

В качестве примера рассмотрим индексатор, который позволяет получить n-член последовательности Фиббоначи:

class DemoFib

{

public int this[int i] //индексатор, доступный только для чтения

{

get

{

if (i <=0) throw new Exception("недопустимое значение индекса");

else if (i==1 || i==2) return 1;

else

{int a=1, b=1, c;

for (int j=3; j<=i; ++j)

{

c=a+b;

a=b;

b=c;

}

return b;

}

}

}

}

class Program

{

static void Main()

{

Console.Write("n=");

int n=int.Parse(Console.ReadLine());

DemoFib a=new DemoFib();

try

{

Console.WriteLine("a[{0}]={1}",n,a[n]);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

Индексаторы очень удобно применять для создания специализированных массивов, на работу с которыми накладываются какие-либо ограничения. Рассмотрим в качестве примера класс-массив, значения элементов которого находятся в диапазоне [0, 100]. Кроме того, при доступе к элементу проверяется, не вышел ли индекс за допустимые границы.

class DemoArray

{

int[] MyArray;//закрытый массив

public DemoArray(int size)//конструктор

{

MyArray = new int[size];

}

public int LengthArray //свойство, возвращающее размерность

{

get { return MyArray.Length; }

}

public int this[int i] //индексатор

{

get

{

if (i <0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else return MyArray[i];

}

set

{

if (i <0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else if (value >= 0 && value <= 100) MyArray[i] = value;

else throw new Exception("присваивается недопустимое значение");

}

}

}

class Program

{

static void Main()

{

DemoArray a = new DemoArray(10);

for (int i=0; i<a.LengthArray; i++)

{

a[i] = i \* i; // использование индексатора в режиме записи

Console.Write(a[i]+" ");// использование индексатора в режиме чтения

}

Console.WriteLine();

try

{

//a[10]=100;

//a[0]=200;

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

Язык С# допускает использование многомерных индексаторов. Они применяются для работы с многомерными массивами. Рассмотрим на примере предыдущую задачу при условии, что организуется двумерный массив.

class DemoArray

{

int[,] MyArray;//закрытый массив

int n, m;//закрытые поля: размерность массива

public DemoArray(int sizeN, int sizeM)//конструктор

{

MyArray = new int[sizeN, sizeM];

this.n = sizeN;

this.m = sizeM;

}

public int LengthN //свойство, возвращающее количество строк

{

get { return n; }

}

public int LengthM //свойство, возвращающее количество строк

{

get { return m; }

}

public int this[int i, int j] //индексатор

{

get

{

if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");

else return MyArray[i, j];

}

set

{

if (i < 0 || i >= n || j < 0 || j >= m) throw new Exception("выход за границы массива");

else if (value >= 0 && value <= 100) MyArray[i, j] = value;

else throw new Exception("присваивается недопустимое значение");

}

}

}

class Program

{

static void Main()

{

DemoArray a = new DemoArray(3, 3);

for (int i = 0; i < a.LengthN; i++,Console.WriteLine())

{

for (int j = 0; j < a.LengthM; j++)

{

a[i, j] = i \*j; // использование индексатора в режиме записи

Console.Write("{0,5}", a[i, j]);// использование индексатора в режиме чтения

}

}

Console.WriteLine();

try

{

//Console.WriteLine(a[3,3]);

//a[0,0]=200;

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

### Операции класса

С# позволяет переопределить большинство операций так, чтобы при использовании их объектами конкретного класса выполнялись действия, отличные от стандартных. Это дает возможность применять объекты собственных типов данных в составе выражений, например:

newObject x, y, z;

...

z = x+y; // используется операция сложения, переопределенная для класса newObject

*Определение* собственных операций класса называют *перегрузкой операций*. *Перегрузка операций* обычно применяется для классов, для которых *семантика* операций делает программу более понятной. Если назначение *операции* интуитивно непонятно, перегружать такую операцию не рекомендуется.

*Операции* класса описываются с помощью методов специального вида, *синтаксис* которых выглядит следующим образом:

[ атрибуты] спецификаторы объявитель\_операции

{тело}

В качестве спецификаторов одновременно используются ключевые слова public и static. Кроме того, операцию можно объявить как внешнюю - extern. Объявление *операции* может выглядеть по-разному, в зависимости от того, что мы перегружаем: унарную или бинарную операцию.

При описании операций необходимо соблюдать следующие правила:

1. операция должна быть описана как открытый статический метод класса ( public static );
2. параметры в операцию должны передаваться по значению (то есть недопустимо использовать параметры ref и out );
3. сигнатуры всех операций класса должны различаться;
4. типы, используемые в операции, должны иметь не меньшие права доступа, чем сама операция (то есть должны быть доступны при использовании операции).

#### Унарные операции

В классе можно переопределять следующие унарные операции: + - ! ~ ++ --, а также константы true и false. При этом, если была перегружена константа true, то должна быть перегружена и константа false, и наоборот.

Синтаксис объявителя унарной операции:

тип operator унарная\_операция (параметр)

Примеры заголовков унарных операций:

public static int operator + (DemoArray m)

public static DemoArray operator --(DemoArray m)

public static bool operator true (DemoArray m)

Параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется. При этом операция должна возвращать:

1. для операций +, -, !, ~ величину любого типа;
2. для операций ++, -- величину типа класса, для которого она определяется;
3. для операций true и false величину типа bool.

Операции не должны изменять значение передаваемого им операнда. Операция, возвращающая величину типа класса, для которого она определяется, должна создать новый объект этого класса, выполнить с ним необходимые действия и передать его в качестве результата.

В качестве примера рассмотрим класс DemoArray, реализующий одномерный массив, в котором содержатся следующие функциональные элементы:

1. конструктор, позволяющий создать объект-массив заданной размерности;
2. конструктор, позволяющий инициализировать объект-массив обычным массивом;
3. свойство, возвращающее размерность массива;
4. индексатор, позволяющий просматривать и устанавливать значение по индексу в закрытом поле-массиве;
5. метод вывода закрытого поля-массива;
6. *перегрузка операции* унарный минус (все элементы массива меняют свое значение на противоположное);
7. *перегрузка операции* инкремента (все элементы массива увеличивают свое значение на 1);
8. перегруза констант true и false (при обращении к объекту будет возвращаться значение true, если все элементы массива положительные, в противном случае, будет возвращаться значение false).

class DemoArray

{

int[] MyArray;//закрытый массив

public DemoArray(int size)//конструктор 1

{

MyArray = new int[size];

}

public DemoArray(params int[] arr)//конструктор 2

{

MyArray = new int[arr.Length];

for (int i = 0; i < MyArray.Length; i++) MyArray[i] = arr[i];

}

public int LengthArray //свойство, возвращающее размерность

{

get { return MyArray.Length; }

}

public int this[int i] //индексатор

{

get

{

if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

return MyArray[i];

}

set

{

if (i < 0 || i >= MyArray.Length) throw new Exception("выход за границы массива");

else MyArray[i] = value;

}

}

public static DemoArray operator -(DemoArray x) //перегрузка операции унарный минус

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = -x[i];

return temp;

}

public static DemoArray operator ++(DemoArray x) //перегрузка операции инкремента

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = x[i]+1;

return temp;

}

public static bool operator true(DemoArray a) //перегрузка константы true

{

foreach (int i in a.MyArray)

{

if (i<0)

{

return false;

}

}

return true;

}

public static bool operator false(DemoArray a)//перегрузка константы false

{

foreach (int i in a.MyArray)

{

if (i>0)

{

return true;

}

}

return false;

}

public void Print(string name) //метод - выводит поле-массив на экран

{

Console.WriteLine(name + ": ");

for (int i = 0; i < MyArray.Length; i++)

Console.Write(MyArray[i] + " ");

Console.WriteLine();

}

}

class Program

{

static void Main()

{

try

{

DemoArray Mas = new DemoArray(1, -4, 3, -5, 0); //вызов конструктора 2

Mas.Print("Исходный массив");

Console.WriteLine("\nУнарный минус");

DemoArray newMas=-Mas; //применение операции унарного минуса

Mas.Print("Mассив Mas"); //обратите внимание, что создается новый объект и знаки меняются

newMas.Print("Массив newMas"); //только у нового массива

Console.WriteLine("\nОперация префиксного инкремента");

DemoArray Mas1=++Mas;

Mas.Print("Mассив Mas");

Mas1.Print("Mассив Mas1=++Mas");

Console.WriteLine("\nОперация постфиксного инкремента");

DemoArray Mas2=Mas++;

Mas.Print("Mассив Mas");

Mas2.Print("Mассив Mas2=Mas++");

if (Mas)

Console.WriteLine("\nВ массиве все элементы положительные\n");

else Console.WriteLine("\nВ массиве есть не положительные элементы\n");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

#### Бинарные операции

При разработке класса можно перегрузить следующие бинарные операции: + - \* / % & | ^ << >> == != < > <= >=. Обратите внимание, операций присваивания в этом списке нет.

Синтаксис объявителя бинарной операции:

тип operator бинарная\_операция (параметр1, параметр 2)

Примеры заголовков бинарных операций:

public static DemoArray operator + (DemoArray a, DemoArray b)

public static bool operator == (DemoArray a, DemoArray b)

При переопределении бинарных операций нужно учитывать следующие правила:

1. Хотя бы один параметр, передаваемый в операцию, должен иметь тип класса, для которого она определяется.
2. Операция может возвращать величину любого типа.
3. Операции отношений определяются только парами и обычно возвращают логическое значение. Чаще всего переопределяются операции сравнения на равенство и неравенство для того, чтобы обеспечить сравнение значения некоторых полей объектов, а не ссылок на объект. Для того чтобы переопределить операции отношений, требуется знание стандартных интерфейсов, которые будут рассматриваться чуть позже.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него две версии *переопределенной операции* +:

* Вариант 1: добавляет к каждому элементу массива заданное число;
* Вариант 2: поэлементно складывает два массива

class DemoArray

{

...

public static DemoArray operator +(DemoArray x, int a) //вариант 1

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i]=x[i]+a;

return temp;

}

public static DemoArray operator +(DemoArray x, DemoArray y) //вариант 2

{

if (x.LengthArray == y.LengthArray)

{

DemoArray temp = new DemoArray(x.LengthArray);

for (int i = 0; i < x.LengthArray; ++i)

temp[i] = x[i] + y[i];

return temp;

}

else throw new Exception("несоответствие размерностей");

}

}

class Program

{

static void Main()

{

try

{

DemoArray a = new DemoArray(1, -4, 3, -5, 0);

a.Print("Массива a");

DemoArray b=a+10;

b.Print("\nМассива b");

DemoArray c = a+b;

c.Print("\nМассива c");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}

#### Операции преобразования типов

Операции преобразования типов обеспечивают возможность явного и *неявного преобразования* между *пользовательскими типами данных*. Синтаксис объявителя операции преобразования типов выглядит следующим образом:

explicit operator целевой\_тип (параметр) //явное преобразование

implicit operator целевой\_тип (параметр) //неявное преобразование

Эти операции выполняют преобразование из типа параметра в тип, указанный в заголовке операции. Одним из этих типов должен быть класс, для которого выполняется преобразование.

*Неявное преобразование* выполняется автоматически в следующих ситуациях:

1. при присваивании объекта переменной целевого типа;
2. при использовании объекта в выражении, содержащем переменные целевого типа;
3. при передаче объекта в метод параметра целевого типа;
4. при явном приведении типа.

Явное преобразование выполняется при использовании операции приведения типа.

При определении операции преобразования типа следует учитывать следующие особенности:

1. тип возвращаемого значения (целевой\_тип) включается в сигнатуру объявителя операции;
2. ключевые слова explicit и *implicit* не включаются в сигнатуру объявителя операции.

Следовательно, для одного и того класса нельзя определить одновременно и явную, и неявную версию. Однако, т.к. *неявное преобразование* автоматически выполнятся при явном использовании операции приведения типа, то достаточно разработать только неявную версию операции преобразования типа.

В качестве примера вернемся к классу DemoArray, реализующему одномерный массив, и добавим в него неявную версию переопределения типа DemoArray в тип одномерный массив и наоборот:

class DemoArray

{

…

public static implicit operator DemoArray (int []a) //неявное преобразование типа int [] в DemoArray

{

return new DemoArray(a);

}

public static implicit operator int [](DemoArray a) //неявное преобразование типа DemoArray в int []

{

int []temp=new int[a.LengthArray];

for (int i = 0; i < a.LengthArray; ++i)

temp[i] = a[i];

return temp;

}

}

class Program

{

static void arrayPrint(string name, int[]a) //метод, который позволяет вывести на экран одномерный массив

{

Console.WriteLine(name + ": ");

for (int i = 0; i < a.Length; i++)

Console.Write(a[i] + " ");

Console.WriteLine();

}

static void Main()

{

try

{

DemoArray a = new DemoArray(1, -4, 3, -5, 0);

int []mas1=a; //неявное преобразование типа DemoArray в int []

int []mas2=(int []) a; //явное преобразование типа DemoArray в int []

DemoArray b1 =mas1; //неявное преобразование типа int [] в DemoArray

DemoArray b2 =(DemoArray)mas2; //явное преобразование типа int [] в DemoArray

//изменение значений

mas1[0]=0; mas2[0]=-1;

b1[0]=100; b2[0]=-100;

//вывод на экран

a.Print("Массива a");

arrayPrint("Maccив mas1", mas1);

arrayPrint("Maccив mas2", mas2);

b1.Print("Массива b1");

b2.Print("Массива b2");

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

}

}