**1. Члены класса:**

| **Член** | **ОПИСАНИЕ** |
| --- | --- |
| [Поля](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/fields) | Поля являются переменными, объявленными в области класса. Поле может иметь встроенный числовой тип или быть экземпляром другого класса. Например, в классе календаря может быть поле, содержащее текущую дату. |
| [Константы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/constants) | Константы — это поля, значения которых устанавливаются во время компиляции и не изменяются. |
| [Свойства](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/properties) | Свойства — это методы класса. Доступ к ним осуществляется так же, как если бы они были полями этого класса. Свойство может защитить поле класса от изменений (независимо от объекта). |
| [Методы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/methods) | Методы определяют действия, которые может выполнить класс. Методы могут принимать параметры, предоставляющие входные данные, и возвращать выходные данные посредством параметров. Методы могут также возвращать значения напрямую, без использования параметров. |
| [События](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/events/index) | События предоставляют другим объектам уведомления о различных случаях, таких как нажатие кнопки или успешное выполнение метода. События определяются и переключаются с помощью делегатов. |
| [Инструкции](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/index) | Перегруженные операторы считаются членами типа. При перегрузке оператора его следует определять как открытый статический метод в типе. Для получения дополнительной информации см. раздел [Перегрузка операторов](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/operator-overloading). |
| [Индексаторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/indexers/index) | Индексаторы позволяют индексировать объекты аналогично массивам. |
| [Конструкторы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/constructors) | Конструкторы — это методы, которые вызываются при создании объекта. Зачастую они используются для инициализации данных объекта. |
| [Методы завершения](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/destructors) | Методы завершения очень редко используются в C#. Они являются методами, вызываемыми средой выполнения, когда объект нужно удалить из памяти. Они обычно применяются для правильной обработки ресурсов, которые должны быть высвобождены. |
| [Вложенные типы](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/nested-types) | Вложенными типами являются типы, объявленные в другом типе. Вложенные типы часто применяются для описания объектов, использующихся только типами, в которых эти объекты находятся. |

**2. & 3. Формат объявления метода. Сигнатура метода.**  
Методы объявляются в классе или в структуре путем указания уровня доступа, такого как public или private, необязательных модификаторов, таких как abstract или sealed, возвращаемого значения, имени метода и всех параметров этого метода. Все эти части вместе представляют собой сигнатуру метода. Параметры метода заключаются в скобки и разделяются запятыми. Пустые скобки указывают, что параметры методу не требуются.  
Сигнатура складывается из следующих аспектов:

* Имя метода
* Количество параметров
* Типы параметров
* Порядок параметров
* Модификаторы параметров

**4. Что для метода могут определить модификаторы.**  
public  
Методы с модификатором public являются открытыми и как правило, устанавливают значения для поля или возвращают значения поля.  
private  
Модификатор private установлен по умолчанию, поэтому явно его объявлять не обязательно.   
Закрытые методы выполняют какие либо действия внутри класса.  
protected  
К методам с модификатором protected могут получить доступ только члены базового класса, где они определены и члены производного класса.  
new  
Модификатор new явно скрывает метод базового класса, который имеет такое имя, как и метод производного.   
Не желательно создавать имена членов в производном классе, которые соответствуют именам в базовом классе.  
static  
Модификатор static создает статический метод, который принадлежит только типу, а не его объекту.  
virtual /override  
Полиморфизм — это возможность определения для каждого производного класса собственного способа выполнения одного и того же метода, определенным в базовом классе.   
  
Модификатор virtual используется в базовом классе и указывает, что метод может быть переопределен в производном классе.   
  
Модификатор virtual нельзя использовать с модификаторами override, static, abstract, private.   
  
Модификатор override используется в производном классе и указывает новую реализацию метода, унаследованного от базового класса.  
  
abstract / override  
Абстрактный метод создается с помощью модификатора abstract в абстрактном базовом классе.   
Он не имеет тела.   
Абстрактный метод может быть объявлен только в абстрактном классе.   
Реализация абстрактного метода должна происходить в производном классе.  
extern  
Если код реализации метода написан не на языке C#, то такой метод называют внешним.   
При объявлении внешнего метода используется модификатор extern, так же нужно указать в объявлении, что такой метод должен быть открытым public и статическим static.  
  
**5. Оператор return.**  
Оператор return завершает выполнение метода, в котором он присутствует, и возвращает управление вызывавшему методу. Он также возвращает необязательное значение. Если метод имеет тип void, оператор return можно опустить.  
Если оператор return находится внутри блока try, блок finally, если он существует, будет выполняться до возврата управления вызывающему методу.  
  
**6. Способ передачи данных между объектами.**  
1) Создание нового статического класса, который будет связан с ними.  
2) Создадим два класса - Form1 и SecondClass,  
в двух различных именованных пространствах - First и Second.  
При инициализации первичного класса  
запускается SecondMetod во вторичном классе,  
который переустанавливает значение переменной FirstString в первичном классе,   
а затем возвращает это значение своей переменной.  
  
**7. Локальные и глобальные переменные.**

**Локальная переменная** - переменная, объявленная внутри какой-либо функции. Областью видимости локальных переменных является тело функции, в которой эта переменная объявлена. Локальная переменная может быть проинициализирована константой или выражением, соответствующими ее типу. (Определенные на уровне метода, доступны только в рамках данного метода. В других методах они недоступны)

**Глобальная переменная** - переменная, объявленная за пределами всех функций. Областью видимости глобальных переменных является вся программа. Глобальная переменная не локализована ни на каком уровне. Глобальная переменная может быть проинициализирована только соответствующей ее типу константой (но не выражением). Инициализация глобальных переменных производится однократно перед началом исполнения специальных функций.

**8.Формальные и фактические параметры.**

Формальные параметры - это собственно параметры метода, а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

**9.Передача параметров по значению и по ссылке.**

**Передача параметров по значению**

Наиболее простой способ передачи параметров представляет передача по значению, по сути это обычный способ передачи параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Program  {      static void Main(string[] args)      {          Sum(10, 15);        // параметры передаются по значению          Console.ReadKey();      }      static int Sum(int x, int y)      {          return x + y;      }  } |

**Передача параметров по ссылке и модификатор ref**

При передаче параметров по ссылке перед параметрами используется модификатор **ref**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | static void Main(string[] args)  {      int x = 10;      int y = 15;      Addition(ref x, y); // вызов метода      Console.WriteLine(x);   // 25        Console.ReadLine();  }  // параметр x передается по ссылке  static void Addition(ref int x, int y)  {      x += y;  } |

Обратите внимание, что модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

Отличие: при передаче по значению метод получает не саму переменную, а ее копию. А при передаче параметра по ссылке метод получает адрес переменной в памяти. И, таким образом, если в методе изменяется значение параметра, передаваемого по ссылке, то также изменяется и значение переменной, которая передается на его место.

**10.Передача констант через параметры.**

Поля для чтения можно инициализировать при их объявлении либо на уровне класса, либо инициилизировать и изменять в конструкторе. Инициализировать или изменять их значение в других местах нельзя, можно только считывать их значение.

Поле для чтения объявляется с ключевым словом **readonly.**

Для ссылочных типов readonly не запрещает изменение состояния объекта, но запрещает изменение ссылки на объект.

То есть можем сделать так:

private readonly List<t> myList;

и потом вот так:

void Adding (T t)

{

myList.Add(t);

}

А вот так уже сделать не можем:

void ChangeList (List<t> newlist)

{

this.myList = newlist;

}

**11.Преимущества и недостатки использование глобальных переменных.**

Недостатки:

1) Сложности при работе с переданными параметрами (передача параметров).

2) Вычисления внутри подпрограммы

Неопытный программист может сказать, что общедоступность глобальных переменных - это хорошо: "ведь это простой способ передать данные". Но если чуть подумать, то это оказывается не так уж здорово - и вот почему:

1.Масштабирование

2.Побочные эффекты

3.Проблемы инициализации

1. Масштабирование

Это просто. Когда вы используете глобальные переменные для чего бы то ни было, вы тем самым неявно предполагаете, что это "что-то" может быть только в одном экземпляре.

2. Побочные эффекты

Вторая проблема - это контроль за изменениями в глобальных переменных. Дело в том, что глобальные переменные видимы отовсюду, глобально. Это удобно: ведь нет никаких ограничителей. С другой стороны, становится совершенно невозможно отследить, кто меняет данные. Неконтролируемые изменения - это первое, что обычно приходит в голову на вопрос о том, чем же плохи глобальные переменные.

3. Проблемы инициализации

Здесь я скажу совсем кратко: очень легко проморгать момент, что для вызова чего-то вам нужно предварительно инициализировать какую-то глобальную переменную. Откуда следуют всевозможные проблемы перекрытия жизненных циклов.

3) Тестирование

Если говорить научно, то глобальные переменные увеличивают число зависимостей между компонентами. Модульный тест - это просто один из примеров на практике, где этот момент хорошо виден.

4) Синглтоны

Преимущества:

1) Глобальные константы

Ещё один случай, когда глобальные переменные обычно оправданы - это, так называемые, переменные-константы.

2) Переменная доступна из любого места в коде.

**12.Модификаторы доступа.**

* public: публичный, общедоступный класс или член класса. Такой член класса доступен из любого места в коде, а также из других программ и сборок.
* private: закрытый класс или член класса. Представляет полную противоположность модификатору public. Такой закрытый класс или член класса доступен только из кода в том же классе или контексте.
* protected: такой член класса доступен из любого места в текущем классе или в производных классах. При этом производные классы могут располагаться в других сборках.
* internal: класс и члены класса с подобным модификатором доступны из любого места кода в той же сборке, однако он недоступен для других программ и сборок (как в случае с модификатором public).
* protected internal: совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.
* private protected: такой член класса доступен из любого места в текущем классе или в производных классах, которые определены в той же сборке.

**13.Время существования локальных и глобальных переменных.**

Гл. - Их область действия распространяется, по крайней мере, на весь модуль. Глобальные переменные играют важную роль, поскольку они обеспечивают весьма эффективный способ обмена информацией между различными частями модуля. Обратная сторона эффективности аппарата глобальных переменных - их опасность.

На самом деле, ситуация с процедурным блоком в C# не так проста. Процедурный блок имеет сложную структуру; в него могут быть вложены другие блоки, связанные с операторами выбора, цикла и так далее. В каждом таком блоке, в свою очередь, допустимы вложения блоков. В каждом внутреннем блоке допустимы объявления переменных. Переменные, объявленные во внутренних блоках, локализованы именно в этих блоках, их область видимости и время жизни определяются этими блоками. Локальные переменные, объявленные в любом внутреннем блоке, существуют от точки объявления до конца соответствующего блока.

**14.Области видимости локальной переменной.**

Область видимости, или контекст переменной — это часть кода, в пределах которого доступна данная переменная.

Локальная переменная находится в области видимости до тех пор, пока закрывающая фигурная скобка не укажет конец блока операторов или метода, в котором она объявлена.

**15.Поля This и Super**

This и super - это два специальных ключевых слова в Java, которые представляют соответственно текущий экземпляр класса и его суперкласса. this представляет текущий экземпляр класса, в то время как super - текущий экземпляр родительского класса.

Справочник по C#. Нет Super. Ключевое слово this ссылается на текущий экземпляр класса, а также используется в качестве модификатора первого параметра метода расширения.

**16.Перекрытие области видимости.**

Область видимости в C# определяются следующими правилами: Поле или переменная – член класса находится в области видимости до тех пор, пока в этой области находится содержащий это поле класс; Локальная переменная находится в области видимости до тех пор, пока закрывающая фигурная скобка не укажет конец блока операторов или метода, в котором она объявлена;

Перекрытие идентификаторов 23 Имена локальных переменных могут перекрывать область видимости одноименных глобальных переменных. При этом все операции с объектом данных под данным именем будут осуществляться со значением локальной переменной.

**17.Статические члены класса**.

Статический член вызывается для класса даже в том случае, если не создан экземпляр класса. Доступ к статическому члены всегда выполняется по имени класса, а не экземпляра. Существует только одна копия статического члена, независимо от того, сколько создано экземпляров класса. Статические методы и свойства не могут обращаться к нестатическим полям и событиям в их содержащем типе, и они не могут обращаться к переменной экземпляра объекта, если он не передается явно в параметре метода.

**18.Инициализация значения статического поля.**

Дело в том, что статические поля не могут рассматриваться как неинициализированные значения. Если они доступны, прежде чем им было присвоено значение, то возвращается то значение, которое используется по умолчанию для этого типа данных. Для целочисленного значения это нуль. Также спецификация языка C # говорит нам о том, что, когда статические поля инициализируются путем применения значения в их декларации, как мы делали выше, они устанавливаются в том порядке, в котором они появляются в коде.

**19.Объявление статического метода.**

С помощью модификатора static.

Следующий класс объявляется как static и содержит только методы static:

static class CompanyEmployee

{

public static void DoSomething() { /\*...\*/ }

public static void DoSomethingElse() { /\*...\*/ }

}

20.**Вызов статического поля и метода**.

Для вызова статического метода не нужен объект. Статический метод позволяет вызывать метод, не имея в наличии ни одного объекта. Вместо имени объекта указывается имя класса.

**21.Передача ссылочных типов в методы.**

Переменная ссылочного типа содержит не сами данные, а ссылку на них. При передаче параметра ссылочного типа по значению можно изменять данные, относящиеся к объекту, на который указывает ссылка, например, значение члена класса. Тем не менее вы не можете изменить значение самой ссылки. Например, вы не можете использовать одну и ту же ссылку, чтобы выделить память для нового объекта и сохранить его вне этого метода. Для этого необходимо передать параметр с использованием ключевого слова ref или out.

**22.Понятия «ссылочная переменная» и «объект, связанный со ссылочной переменной».**

Ссылочные же переменные хранят лишь ссылки на объекты, причем различные переменные могут ссылаться на один и тот же объект. То есть объектные переменные после приравнивания остаются "связанными" друг с другом, изменения одной сказываются на другой. Point p 1 = new Point(3, 5); Point p 2=p 1; p 1. x=7; print(p 2. x);

**23.Передача строк по ссылке**.

String это особенный ссылочный тип. Любая операция меняющую строку возвращает модифицированную строку не меняя оригинал.

Всё верно - строки - неизменяемы и являются ссылочными типами, т.е. передаются по ссылке.

Если вы присваивается это значение, сохраняется ссылка на новый участок памяти. Старая строка становится объектом интереса сборщика мусора.

Если хотите изменять строку внутри метода, используйте ref-параметры.

**24.Варианты отношений между классами.**

1. Наследование

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением.

2. Реализация

Реализация предполагает определение интерфейса и его реализация в классах.

3. Ассоциация

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа.

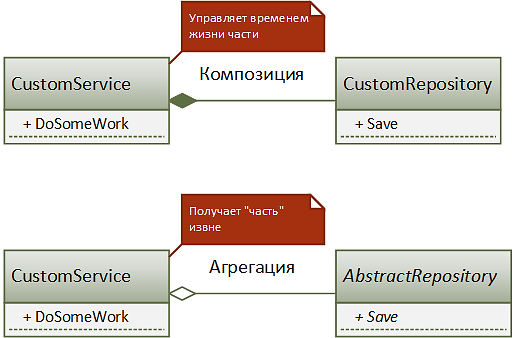
4. Композиция

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя.

5. Агрегация

От композиции следует отличать агрегацию. Она также предполагает отношение HAS A, но реализуется она иначе.

**25.****Отношение агрегирования между классами.**



При агрегации реализуется слабая связь.

при использовании агрегации, объект-содержимое может принадлежать даже нескольким контейнерам (но тут нужно быть осторожным) или у одного объекта-контейнера, в течении «жизни», может быть разное содержимое (причем, агрегация позволяет использовать даже содержимое разных типов одном у тому же контейнеру).

На самом деле, можно сказать, что композиция — это частный, более строгий, вариант агрегации.

**26.Отношение зависимости между классами.**

Два основных типа отношений между классами определены в ОО-системах. Первое отношение "клиенты и поставщики", называется часто клиентским отношением или отношением вложенности (встраивания). Второе отношение "родители и наследники" называется отношением наследования.

Определение 1. Классы А и В находятся в отношении "клиент-поставщик", если одним из полей класса В является объект класса А. Класс А называется поставщиком класса В, класс В называется клиентом класса А.

Определение 2. Классы А и В находятся в отношении "родитель - наследник", если при объявлении класса В класс А указан в качестве родительского класса. Класс А называется родителем класса В, класс В называется наследником класса А.

Мощь ООП основана на *наследовании*. Когда построен полезный класс, то он может многократно использоваться. Повторное использование - это одна из главных целей ООП. Но и для хороших классов неизбежно наступает момент, когда необходимо расширить возможности класса, придать ему новую функциональность, изменить интерфейс.

**27.Внутренний класс.**

Внутренний класс ведет себя как обычный класс за тем исключением, что его объекты могут быть созданы только внутри внешнего класса.

Внутренний класс имеет доступ ко всем полям внешнего класса, в том числе закрытым с помощью модификатора private. Аналогично внешний класс имеет доступ ко всем членам внутреннего класса, в том числе к полям и методам с модификатором private.

**28.Понятие конструктора класса.**

Вызов конструктора - потомка приводит к цепочке вызовов конструкторов - предков, заканчивающейся вызовом конструктора прародителя. Затем в обратном порядке создаются объекты, начиная с объекта прародителя, и выполняются тела соответствующих конструкторов, инициализирующие поля и выполняющие другую работу этих конструкторов. Последним создается объект потомка и выполняется тело конструктора потомка.

**29. Пример конструктора класса.**

public Derived() {}

public Derived(string name, int cred, int deb):base (name,cred)

{

debet = deb;

}

**30.Анонимный объект.**

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова var и инициализатора объектов:

1 2 var user = new { Name = "Tom", Age = 34 }; Console.WriteLine(user.Name);

В данном случае user - это объект анонимного типа, у которого определены два свойства Name и Age. И мы также можем использовать его свойства, как и у обычных объектов классов. Однако тут есть ограничение - свойства анонимных типов доступны только для чтения.

Также можно определять массивы объектов анонимных типов:

var people = new[] { new {Name="Tom"}, new {Name="Bob"} }; foreach(var p in people) { Console.WriteLine(p.Name); }

**31.Пример использования анонимного объекта**.

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Анонимный тип определяется с помощью ключевого слова var и инициализатора объектов:

1. 1 var user = new { Name = "Tom", Age = 34 };

2. 2. Console.WriteLine(user.Name)

**32.Наследование.**

Наследование (inheritance) является одним из ключевых моментов ООП. Благодаря наследованию один класс может унаследовать функциональность другого класса.

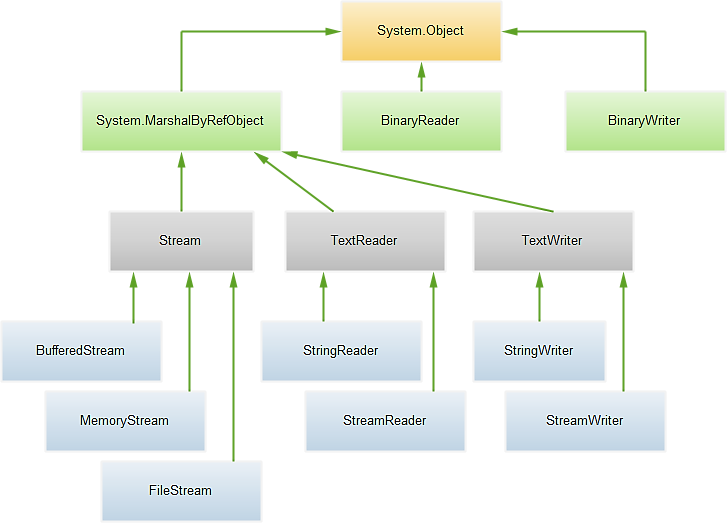
При создании потомка – после двоеточия мы указываем базовый класс для данного класса.

**33.Формат задания класса-потомка.**

При задании класса потомка сначала идут модификаторы, затем после ключевого слова class идёт имя декларируемого класса, потом обязательно ставится «:», после чего требуется указать имя класса родителя.

**34.Иерархия классов.**

Создание многоуровневой иерархии классов возможно с помощью наследования. Например, один основной класс, от него наследуются допустим два, от каждого из этих еще по три и так далее.



**35.Терминальные классы и методы.**

Терминальный класс (final class) — класс, объявленный с ключевым словом final, обозначающим, что он не может использоваться в качестве базового при определении новых классов и наследовании.

Терминальный метод (final method) — метод, объявленный с ключевым словом final, обозначающим, что он не может переопределяться в подклассах.

**36.Переопределение методов.**

(Виртуальные методы) А чтобы переопределить метод в классе-наследнике, этот метод определяется с модификатором override. Переопределенный метод в класе-наследнике должен иметь тот же набор параметров, что и виртуальный метод в базовом классе.

**37.Перегрузка методов.**

Перегрузка метода как раз заключается в том, что методы имеют разную сигнатуру, в которой совпадает только название метода. То есть методы должны отличаться по:

* Количеству параметров
* Типу параметров
* Порядку параметров
* Модификаторам параметров

**38.Абстрактные классы.**

Абстрактный класс похож на обычный класс. Он также может иметь переменные, методы, конструкторы, свойства. Единственное, что при определении абстрактных классов используется ключевое слово abstract.

Но главное отличие состоит в том, что мы не можем использовать конструктор абстрактного класса для создания его объекта.

Однако несмотря на то, что напрямую мы не можем вызвать конструктор класса Person для создания объекта, тем не менее конструктор в абстрактных классах то же может играть важную роль, в частности, инициализировать некоторые общие для производных классов переменные и свойства.

**39.Формат объявления абстрактного метода.**

Абстрактный метод – это метод, который не имеет своей реализации в базовом классе, и он ДОЛЖЕН быть реализован в классе-наследнике. Абстрактный метод может быть объявлен только в абстрактном классе

Абстрактный метод должен быть реализован в классе наследнике, виртуальный метод переопределять необязательно.

[модификатор доступа] abstract [тип] [имя метода] ([аргументы]);

**40.Наследование закрытых членов класса.**

Закрытые члены являются видимыми только в производных классах, которые вложены в базовый класс. Для других производных классов они невидимы.

При закрытом наследовании все члены родительского класса наследуются как закрытые. Это означает, что члены private остаются недоступными, а члены protected и public становятся private в дочернем классе.

**41.Защищённые члены класса.**

Члены класса (включая вложенные классы и структуры) можно объявлять с любым из шести типов доступа. Члены структуры нельзя объявлять как защищенные, поскольку структуры не поддерживают наследование.

Защищенный внутренний уровень доступности означает защищенный ИЛИ внутренний доступ, а не защищенный И внутренний. Другими словами, защищенный внутренний член доступен из любого класса в той же сборке, включая производные классы. Чтобы сделать его доступным только для производных классов в той же сборке, объявите сам класс как внутренний, а его члены как защищенные

**42.Методы set и get.**

Стандартное определение свойства содержит блоки get и set. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.

**43.Класс Object.**

Все остальные классы в .NET, даже те, которые мы сами создаем, а также базовые типы, такие как System.Int32, являются неявно производными от класса Object. Даже если мы не указываем класс Object в качестве базового, по умолчанию неявно класс Object все равно стоит на вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе System.Object.

**44.Методы класса Object.**

ToString

Метод ToString служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение.

Метод GetHashCode

Метод GetHashCode позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код.

Получение типа объекта и метод GetType

Метод GetType позволяет получить тип данного объекта.

Метод Equals

Метод Equals позволяет сравнить два объекта на равенство.

Clone()

Этот метод создает копию объекта и возвращает ссылку на эту копию (а в случае типа значения — ссылку на упаковку). при этом выполняется неглубокое копирование.

**45.Статическое связывание методов.**

Канонический пример связывания — сопоставление имени с конкретной функцией при компиляции выражения.

Независимо от того, какой метод будет привязан, суть состоит в том, что связывание будет завершено компилятором. При этом связывание полностью зависит от того, известны ли на момент компиляции все необходимые типы. Такое связывание называется статическим.

**46.Динамическое связывание методов.**

Конкретный вариант вызываемого виртуального метода непосредственно в момент вызова (в противоположность статическому связыванию — определению метода на момент компиляции).

**47.Два подхода к определению полиморфизма.**

Полиморфизм может быть статическим или динамическим. В статическом полиморфизме, реакция на функцию, определяется во время компиляции. В динамическом полиморфизме, решается во время работы (run-time).

[ Полиморфизм в языках программирования и теории типов — способность функции обрабатывать данные разных типов. Существует два похода:

1. Параметрический полиморфизм в языках программирования и теории типов — свойство семантики системы типов, позволяющее обрабатывать значения разных типов идентичным образом, то есть исполнять физически один и тот же код для данных разных типов.

2. Ad-hoc-полиморфизм поддерживается во многих языках посредством перегрузки функций и методов, а в слабо типизированных — также посредством приведения типов. ]

**48.Динамические и виртуальные методы.**

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес.Отсюда и достоинства и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса только тех динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, TObject, где имеется стандартный обработчик вызова динамических методов.

Таким образом, различие виртуальных и динамических методов состоит только в способе хранения из адресов. Адрес виртуального метода дальнего предка можно быстро взять из таблицы VMT данного класса, а для поиска адреса динамического метода дальнего предка придется по очереди перебирать всю иерархическую цепочку данного класса, что гораздо медленнее.

Директива override указывает, что перекрывается динамический или виртуальный метод класса-предка. Без этой директивы перекрытие перекрывается динамических или виртуальных методов невозможно.

**49. Понятие массива.**

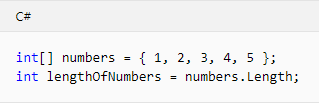
Массив — это пронумерованная последовательность величин одинакового типа, обозначаемая одним именем.

**50. Статические и динамические массивы.**

Статистическими(традиционными) массивами, называются те массивы при объявлении которых в явном или косвенном виде указываются конкретные границы изменения каждого индекса. Динамический массив, в отличии от статического, может изменять свой размер.

# **51. Массивы как объекты.**

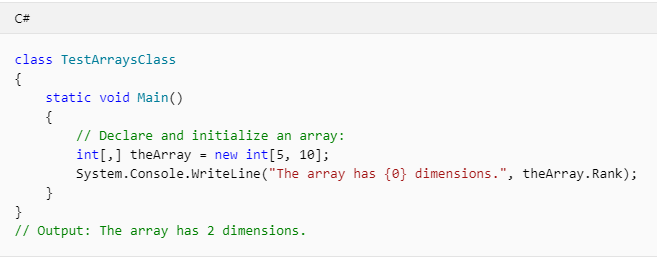
В C# массивы представляют собой реальные объекты, а не просто адресуемые области непрерывной памяти, как в C и C++.[Array](https://docs.microsoft.com/ru-ru/dotnet/api/system.array) — это абстрактный базовый тип для всех типов массивов. Вы можете использовать свойства и другие члены класса, входящие в [Array](https://docs.microsoft.com/ru-ru/dotnet/api/system.array). Например, с помощью свойства [Length](https://docs.microsoft.com/ru-ru/dotnet/api/system.array.length) можно получить длину массива. В следующем коде значение длины массива numbers (5) присваивается переменной с именем lengthOfNumbers:



В классе [Array](https://docs.microsoft.com/ru-ru/dotnet/api/system.array) представлено множество других полезных методов и свойств для сортировки, поиска и копирования массивов.

## Пример

В этом примере используется свойство [Rank](https://docs.microsoft.com/ru-ru/dotnet/api/system.array.rank), позволяющее отобразить число измерений массива.



**52. Создание и инициализация одномерного массива.**

Массив представляет набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки.

Вы можете объявить одномерный массив, содержащий пять целых чисел, как показано в следующем примере:

int[] array = new int[5];

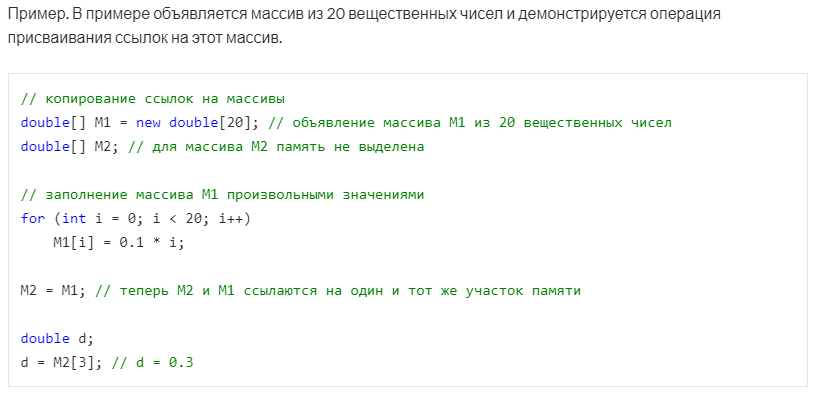
Массив можно инициализировать при объявлении. В этом случае не требуется спецификатор длины, поскольку он уже задан по числу элементов в списке инициализации. Например:

int[] array1 = new int[] { 1, 3, 5, 7, 9 };

**53. Присваивание массивов.**

Для любого новосозданного массива память выделяется с помощью оператора new. Это означает, что массивы есть ссылочного типа (а не типа значения). То есть, переменные типа «массив» есть ссылками на эти массивы.  
После присваивания значения одной переменной (ссылки на массив) другой переменной, обе переменные будут указывать (ссылаться) на один и тот же массив. То есть, копия массива не создается.

Чтобы переменные указывали на разные участки памяти (разные массивы данных) – для любой из них нужно выделять память оператором new. Если нужно выполнить копирование данных одного массива в другой, то это нужно выполнить программным путем, например с помощью операторов цикла.

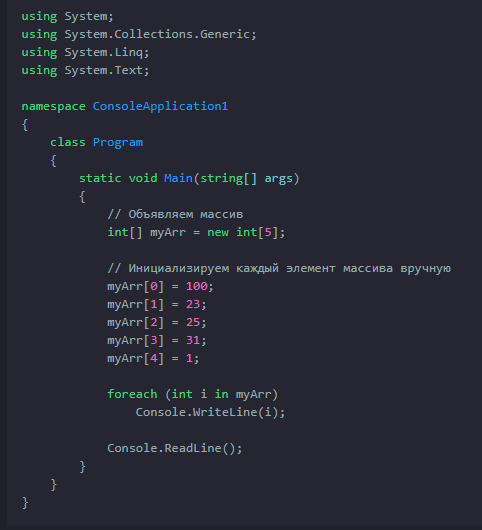


**54. Массивы объектов. Пример.**

**Массив** представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В C# массивы могут быть как одномерными, так и многомерными. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных.

Массивами в C# можно пользоваться практически так же, как и в других языках программирования. Тем не менее у них имеется одна особенность: они реализованы в виде объектов.

Для тoго чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор new.



**55. Формат объявления и инициализация многомерного массива**.

Многомерным называется такой массив, который отличается двумя или более измерениями, причем доступ к каждому элементу такого массива осуществляется с помощью определенной комбинации двух или более индексов. Многомерный массив индексируется двумя и более целыми числами.

Простейшей формой многомерного массива является двумерный массив. Местоположение любого элемента в двумерном массиве обозначается двумя индексами. Такой массив можно представить в виде таблицы, на строки которой указывает один индекс, а на столбцы — другой. Пример объявления и инициализации двумерного массива показан ниже:

// Объявляем двумерный массив

int[,] myArr = new int[4, 5];

Random ran = new Random();

// Инициализируем данный массив

for (int i = 0; i < 4; i++)

{

for (int j = 0; j < 5; j++)

{

myArr[i, j] = ran.Next(1, 15);

Console.Write("{0}\t", myArr[i, j]);

}

Console.WriteLine();

}

**56. Регулярные и иррегулярные массивы.**

Для того, чтобы работать с регулярными выражениями необходимо подключить в начале программы пространство имен using System.Text.RegularExpressions;В Си-шарп работу с регулярными выражениями предоставляет класс Regex. Создание регулярного выражения имеет следующий вид:

Regex myReg = new Regex([шаблон]);

Здесь [шаблон] – это строка содержащая символы и спецсимволы. У Regex также есть и второй конструктор, который принимает дополнительный параметр – опции поиска. Пример: static void Main(string[] args) { string data1 = "Петр, Андрей, Николай"; string data2 = "Петр, Андрей, Александр"; Regex myReg = new Regex("Николай"); // создание регулярного выражения Console.WriteLine(myReg.IsMatch(data1)); // True Console.WriteLine(myReg.IsMatch(data2)); // False Console.ReadKey(); }

Здесь в качестве шаблона выступает однозначная строка "Николай". Дальше был использован метод IsMatch, который проверят, содержит ли заданная строка (data1, data2) подстроку соответствующую шаблону.

**57. Клонирование и копирование массивов.**

Для копирования массивов предусмотрена реализация массивами интерфейса ICloneable. Метод Clone(), определенный в этом интерфейсе, создает неглубокую копию массива. Если элементы

массива относятся к типу значений, то все они копируются, если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них.

Вместо метода Clone() можно также применять метод Array.Сору(), тоже создающий поверхностную копию. Но между Clone() и Сору() есть одно важное отличие: Clone() создает новый массив, а Сору() требует наличия существующего массива той же размерности с достаточным количеством элементов.

**58. Сравнение массивов.**

LINQ предоставляет встроенную функцию для проверки равенства двух IEnumerable s, и эта функция может использоваться на массивах.

Функция [SequenceEqual](https://msdn.microsoft.com/en-us/library/bb348567(v=vs.110).aspx" \t "_blank) вернет true если массивы имеют одинаковую длину, а значения в соответствующих индексах равны, а false противном случае.

**59. Особенности символьных массивов**.

Особенности скорее связаны со спецификой типа char, чем со структурой или свойствами непосредственно массива. Символьный массив создается стандартным способом: одновременно с объявлением переменной массива words списком символов инициализируются элементы массива. В результате выполнения команды System.out.println(words) на экран выводится сообщение Символьный массив.

**60. Понятие коллекции.**

Коллекция в программировании — программный объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов, и позволяющий обращаться к этим значениям. Коллекции - это массивы, вектора, матрицы, стеки, списки, множества и т. д.

**61. Элементы коллекции.**

Основой для создания всех коллекций является реализация интерфейсов IEnumerator и IEnumerable (и их обобщенных двойников IEnumerator<T> и IEnumerable<T>). Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле foreach. А интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable (IEnumerable<T>) является базовым для всех коллекций.

$collection = new Collection;

$collection->add('Элемент №1');

$collection->add('Элемент №2');

$collection->add('Элемент №3');

echo 'Записей в коллекции: ', $collection->count();

**62. Пример создания списка на основе интерфейса-коллекции.**

// Create a list of strings.

var salmons = new List<string>();

salmons.Add("chinook");

salmons.Add("coho");

salmons.Add("pink");

salmons.Add("sockeye");

// Iterate through the list.

foreach (var salmon in salmons)

{

Console.Write(salmon + " ");

}

// Output: chinook coho pink sockeye

**63. Основные классы и интерфейсы коллекций.**

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, коллекция, с которой вы работаете, может расти или уменьшаться динамически при необходимости. Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции.

Коллекция является классом, поэтому необходимо объявить экземпляр класса перед добавлением в коллекцию элементов.

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен System.Collections.Generic. Универсальная коллекция обеспечивает строгую типизацию, так что в нее нельзя добавить другие типы данных. При извлечении элемента из универсальной коллекции не нужно определять или преобразовывать его тип данных.

**64.Понятие итератора.**

Итератор по сути представляет блок кода, который использует оператор yield для перебора набора значений. Данный блок кода может представлять тело метода, оператора или блок get в свойства.

Итератор использует две специальных инструкции:

yield return: определяет возвращаемый элемент

yield break: указывает, что последовательность больше не имеет элементов

**65. Пример создания и использования итератора.**

Итератор - это указатель на элемент контейнера.

Нужен он для того, чтобы работать с элементами контейнера и алгоритмами обобщенно,

независимо от того, какой именно контейнер используется.

Дело в том, что разные контейнеры STL по-разному реализуют доступ к своим элементам.

У vector они расположены последовательно в памяти, у list они связаны через указатели и т.д.

А итератор маскирует эти различия за обобщенным фасадом указателя.

Благодаря итераторам, можно использовать один и тот же алгоритм для разных контейнеров,

не меняя при этом ни строчки кода. Простейший пример:

for (container\_t::iterator it = Con.begin(); it != Con.end(); ++it)

{

// ...

}

**66. Методы итератора.**

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор - интерфейс ListIterator. Данный итератор используется классами, реализующими интерфейс List, то есть классами LinkedList, ArrayList и др.

Интерфейс ListIterator расширяет интерфейс Iterator и определяет ряд дополнительных методов:

· void add(E obj): вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом next()

· boolean hasNext(): возвращает true, если в коллекции имеется следующий элемент, иначе возвращает false

· boolean hasPrevious(): возвращает true, если в коллекции имеется предыдущий элемент, иначе возвращает false

· E next(): возвращает текущий элемент и переходит к следующему, если такого нет, то генерируется исключение NoSuchElementException

· E previous(): возвращает текущий элемент и переходит к предыдущему, если такого нет, то генерируется исключение NoSuchElementException

· int nextIndex(): возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка

· int previousIndex(): возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1

· void remove(): удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов next() или previous(), иначе будет сгенерировано исключение IlligalStateException

· void set(E obj): присваивает текущему элементу, выбранному вызовом методов next() или previous(), ссылку на объект obj

**67. Основные методы класса Collection.**

IEnumerable<T>

ICollection<T>

IList<T>

ISet<T>

IDictionary<TKey, TValue>

ILookup<TKey, TValue>

IComparer<T>

IEqualityComparer<T>

IProducerConsumerCollection<T>

**68. Интерфейс List.**

IList<T> — стандартный интерфейс коллекции поддерживающей индексацию по позиции. Помимо функционала, унаследованного от ICollection<T> и IEnumerable<T>, он предоставляет возможность чтения и записи элемента по позиции (через индексатор) и вставки/удаления по позиции

**69. Методы интерфейса List.**

Класс List<T> представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

* **void Add(T item)**: добавление нового элемента в список
* **void AddRange(ICollection collection)**: добавление в список коллекции или массива
* **int BinarySearch(T item)**: бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
* **int IndexOf(T item)**: возвращает индекс первого вхождения элемента в списке
* **void Insert(int index, T item)**: вставляет элемент item в списке на позицию index
* **bool Remove(T item)**: удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true
* **void RemoveAt(int index)**: удаление элемента по указанному индексу index
* **void Sort()**: сортировка списка

**70. Место интерфейса List в иерархии классов**.

**71. Реализации интерфейса List.**

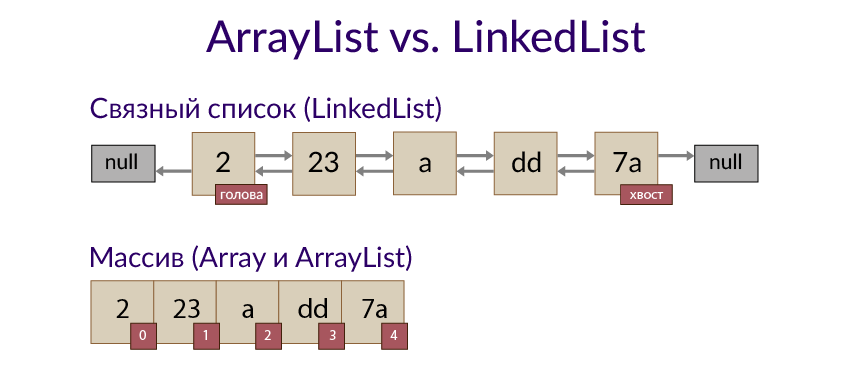
Интерфейс List Список – упорядоченная коллекция У каждого элемента списка (в отличие от обобщенной коллекции) есть своя позиция Могут быть дублированные элементы Добавлены методы, связанные с конкретными позициями Кроме этого, изменены контракты некоторых методов Collection

public interface List extends Collection { E get(int index); int indexOf(Object obj); int lastIndexOf(Object obj); List subList(int fromIndex, int toIndex); // Списочный итератор // Списочный итератор ListIterator listIterator(); // Только для модифицируемых списков // Старые методы add добавляют в конец списка boolean add(int index, E obj); boolean addAll(int index, Collection c); void clear(); E remove(int index); E set(int index, E obj); }

**72. Отличие LinkedList от ArrayList.**

коллекции могут быть реализованы разными способами и нет единственного – самого правильного. При одном подходе одни операции являются быстрыми, а остальные медленными, при другом – все наоборот. Нет одного идеального, подходящего всем решения.

— Поэтому было решено сделать несколько реализаций одной и той же коллекции. И каждая реализация была оптимизирована для какого-то узкого набора операций. Так появились разные коллекции.



**73. Интерфейс Set.**

Интерфейс Set расширяет интерфейс Collection и представляет набор уникальных элементов. Описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Set не добавляет новых методов, только вносит изменения в унаследованные.

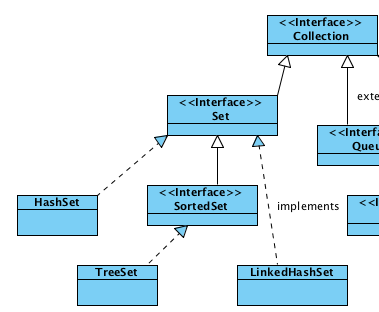
**74. Методы интерфейса Set.**

Использование интерфейсов возможно двумя способами:

* Класс может реализовывать интерфейс. Реализация интерфейса заключается в том, что в описании класса данный интерфейс указывается как реализуемый, а в коде класса обязательно определяются все методы, которые описаны в интерфейсе, в полном соответствии с сигнатурами из описания этого интерфейса. То есть, если класс реализует интерфейс, для любого экземпляра этого класса существуют и могут быть вызваны все описанные в интерфейсе методы. Один класс может реализовать несколько интерфейсов одновременно.
* Возможно объявление переменных и параметров методов как имеющих тип «интерфейс». В такую переменную или параметр может быть записан экземпляр любого класса, реализующего интерфейс. Если интерфейс объявлен как тип возвращаемого значения функции, это означает, что функция возвращает объект класса, реализующего данный интерфейс.

**75. Место интерфейса Set в иерархии классов.**

Интерфейс Collection -> Интерфейс Set - >(Hashset) + (Linkedhashset)+(Sorted set->TreeSet)



**76. Реализации интерфейса Set.**

Интерфейс представляет ссылочный тип, который определяет набор методов и свойств, но не реализует

их. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программировани

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. В данном случае интерфейс определяет метод Move, который будет представлять некоторое передвижение. Он не принимает никаких параметров и ничего не возвращает.

Еще один момент в объявлении интерфейса: все его члены - методы и свойства не имеют модификаторов доступа, но фактически по умолчанию доступ public, так как цель интерфейса - определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

**77. Понятие хэш-функции.**

Хеш-функция или функция свёртки — функция, осуществляющая преобразование массива входных данных произвольной длины в (выходную) битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением». Результат преобразования (выходные данные) называется «хешем», «хеш-кодом», «хеш-суммой», «сводкой сообщения».

Хеш-функции применяются в следующих случаях: при построении ассоциативных массивов;

при поиске дубликатов в сериях наборов данных;

при построении уникальных идентификаторов для наборов данных; при вычислении контрольных сумм от данных (сигнала) для последующего обнаружения в них ошибок (возникших случайно или внесённых намеренно), возникающих при хранении и/или передаче данных;

при сохранении паролей в системах защиты в виде хеш-кода (для восстановления пароля по хеш-коду требуется функция, являющаяся обратной по отношению к использованной хеш-функции); при выработке электронной подписи (на практике часто подписывается не само сообщение, а его «хеш-образ»);

и др. В общем случае (согласно принципу Дирихле) нет однозначного соответствия между хеш-кодом (выходными данными) и исходными (входными) данными. Возвращаемые хеш-функцией значения (выходные данные) менее разнообразны, чем значения входного массива (входные данные). Случай, при котором хеш-функция преобразует несколько разных сообщений в одинаковые сводки, называется «коллизией». Вероятность возникновения коллизий используется для оценки качества хеш-функций.

Существует множество алгоритмов хеширования, отличающихся различными свойствами. Примеры свойств: разрядность; вычислительная сложность; криптостойкость.

Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшим примером хеш-функции может служить «обрамление» данных циклическим избыточным кодом (англ. CRC, cyclic redundancy code)

**78. Особенности класса HashSet.**

HashSet-ы позволяют более оптимально реализовать операции над множествами — пересечение, объединение и вычитание. Однако не стоит забывать об их избыточности. Hash-set-ы резервируют значительно больше памяти, чем нужно для хранения их элементов, поэтому их имеет смысл использовать только для множеств среднего размера (100-10000 элементов). Фактически, вы тратите память, чтобы получить более быстрые вычисления. Для больших множеств следует использовать деревья.

**79. Особенности класса TreeSet.**

Обобщенный класс TreeSet<E> представляет структуру данных в виде дерева, в котором все объекты хранятся в отсортированном виде по возрастанию. TreeSet является наследником класса AbstractSet и реализует интерфейс NavigableSet, а следовательно, и интерфейс SortedSet.

В классе TreeSet определены следующие конструкторы:

· TreeSet(): создает пустое дерево

· TreeSet(Collection<? extends E> col): создает дерево, в которое добавляет все элементы коллекции col

· TreeSet(SortedSet <E> set): создает дерево, в которое добавляет все элементы сортированного набора set

· TreeSet(Comparator<? super E> comparator): создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

**80. Интерфейс Queue**

Класс Queue<T> представляет обычную очередь, работающую по алгоритму FIFO ("первый вошел - первый вышел").

У класса Queue<T> можно отметить следующие методы:

· Dequeue: извлекает и возвращает первый элемент очереди

· Enqueue: добавляет элемент в конец очереди

· Peek: просто возвращает первый элемент из начала очереди без его удаления

**81. Методы интерфейса Queue.**

Интерфейс Queue описывает методы работы с очередями. *Очередью*называется коллекция, элементы в которую добавляются с одного конца, а удаляются с другого конца.

Интерфейс Queue добавляет к методам интерфейса Collection методы, характерные для очередей:

* Object element() — возвращает первый элемент очереди, не удаляя его из очереди. Метод выбрасывает исключение, если очередь пуста;
* Object peek() — возвращает первый элемент очереди, не удаляя его. В отличие от метода element() не выбрасывает исключение;
* Object remove() — возвращает первый элемент очереди и удаляет его из очереди. Метод выбрасывает исключение, если очередь пуста;
* Object poll() — возвращает первый элемент очереди и удаляет его из очереди.
* В отличие от метода remove() не выбрасывает исключение;
* boolean offer(Object obj) — вставляет элемент в конец очереди и возвращает true,

если вставка удалась.

**82. Место интерфейса Queue в иерархии классов.**

**83. Реализации интерфейса Queue.**

public class Queue<T> : System.Collections.Generic.IEnumerable<T>,

**84. Интерфейс Map.**

Интерфейс **Map<K, V>** представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

**85. Объявление интерфейса Map.**

В объявлении интерфейса содержится служебное слово interface, за которым следуют наименование интерфейса и перечень его членов, заключенный в

фигурные скобки.

Map — это структурированные данные, состоящие из набора пар ключ-значение, и каждый ключ может использоваться только один раз в одной Map. Интерфейс Map предлагает три вида коллекций: набор ключей, набор значений и набор ключ-значение.

**86. Методы интерфейса Map.**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| void clear() | Очистка хеш-таблицы |
| boolean containsKey(Object key) | Функция проверки присутствия объекта по ключу |
| boolean containsValue(Object value) | Функция проверки присутствия объекта по значению |
| Set<Map.Entry<K,V>> entrySet() | Функция получения объекта в виде коллекции Set |
| boolean equals(Object object) | Функция сравнения с объектом object |
| Object get(Object key) | Функция получения записи по ключу |
| boolean isEmpty() | Функция проверки наличия записей |
| Set<K> keySet() | Функция получения записей в виде коллекции Set |
| void put(K key, V value) | Функция добавления записи |
| void putAll(Map<? extends K,? extends V> t) | Функция добавления записей |
| void remove(Object key) | Метод удаления объекта по ключу key |
| boolean remove(Object key, Object value) | Функция удаления записи по соответствию значений ключа и значения |
| void replace(K key, V value) | Замена значения value для записи с ключом key |
| boolean replace(K key, V oldValue, V newValue) | Замена значения oldValue на newValue для записи с ключом key |
| int size() | Функция определения количества записей в хеш-таблице |
| Collection<V> values() | Получение значений записей в виде коллекции |

**87. Место интерфейса Map в иерархии классов.**

1. Главным образом для того, чтобы у всех объектов и классов было общее поведение, а также поведение по умолчанию, которое содержится в виртуальных методах, которые можно переопределить. Для этого лучше всего походит единый корневой объект. Это гарантирует, что у всех типов будут определённые методы.

2. Для того, чтобы можно было писать члены (методы, свойства, индексаторы, события и т.д.), которые могут по-разному работать с абсолютно непохожими типами. Однако эта возможность сильно нивелировалась более удобными Generic и удалила необходимость проверки типов и приведений, ошибки в которых можно было увидеть во время исполнения. Однако по-прежнему существует API (в том числе и стандартное), которое было разработано до Generic, и в котором активно используется корневой тип.

**88. Реализации интерфейса Map.**

Интерфейс Map<K, V> представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта. Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений. А метод entrySet() возвращает набор всех элементов в виде объектов Map.Entry<K, V>

Map<Integer, String> states = new HashMap<Integer, String>();

**89. Устаревшие коллекции.**

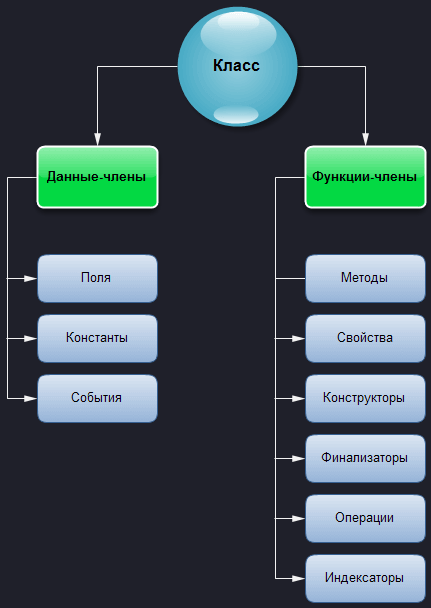
# **90. Классы, реализующие строки в C#**

Для представления текстовой информации в С# используются объекты класса **string.**Класс **string**представляет собой один из предопределённых типов языка С#. В .Net Framework этому типу соответствует класс System.String.

**Общая форма определения класса**

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

Вообще говоря, данные содержатся в членах данных, определяемых классом, а код — в функциях-членах.



**91. Отличие char code unit от char code point.**

code point - это примерно то же, что мы привыкли называть символом. Но не совсем. Например, буква «ё» может быть как одним codepoint'ом, так и двумя - буквой «е» и символом "две точки над предыдущей буквой"

code unit - это единицы кодировки. Байт для utf-8, Слово (два байта) для unf-16 или Длинное слово (четыре байта) для utf-32.

**92. Создание строковых литералов.**

Строковые литералы представляют строки. Строки заключаются в двойные

кавычки:

Console.WriteLine("hello")

Один из видов объектов класса **string**мы уже многократно применяли - это строковые константы или строковые литералы.

**93. Кодировки, поддерживаемые C#.**

Используйте статические свойства класса Encoding, которые возвращают объекты, представляющие стандартные кодировки символов (ASCII, UTF-7, UTF-8, UTF-16 и UTF-32)

**94. Конструкторы строк.**

Инициализирует новый экземпляр класса String.

String(Char\*)

Инициализирует новый экземпляр класса String значением, определенным указателем на массив знаков Юникода.

String(Char[])

Инициализирует новый экземпляр класса String символами Юникода, заданными в виде массива символов.

String(Char, Int32)

Инициализирует новый экземпляр класса String значением, полученным путем повторения заданного знака Юникода указанное число раз.

String(Char\*, Int32, Int32)

Инициализирует новый экземпляр класса String значением, которое определяется заданным указателем на массив знаков Юникода, начальной позицией знака в этом массиве и длиной.

String(Char[], Int32, Int32)

Инициализирует новый экземпляр класса String значением, заданным массивом знаков Юникода, начальной позицией знака в пределах данного массива и длиной.

В языке C# строковые значения представляет тип string, а вся функциональность работы с данным типом сосредоточена в классе **System.String.**

Основная функциональность класса String раскрывается через его методы, среди которых можно выделить следующие:

* **Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal**: сравнивает две строки без учета локали
* **Contains**: определяет, содержится ли подстрока в строке
* **Concat**: соединяет строки
* **CopyTo**: копирует часть строки или всю строку в другую строку
* **EndsWith**: определяет, совпадает ли конец строки с подстрокой
* **Format**: форматирует строку
* **IndexOf**: находит индекс первого вхождения символа или подстроки в строке
* **Insert**: вставляет в строку подстроку
* **Join**: соединяет элементы массива строк
* **LastIndexOf**: находит индекс последнего вхождения символа или подстроки в строке
* **Replace**: замещает в строке символ или подстроку другим символом или подстрокой
* **Split**: разделяет одну строку на массив строк
* **Substring**: извлекает из строки подстроку, начиная с указанной позиции
* **ToLower**: переводит все символы строки в нижний регистр
* **ToUpper**: переводит все символы строки в верхний регистр
* **Trim**: удаляет начальные и конечные пробелы из строки

**95. Особенности конкатенации строк.**

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода Concat:

string s1 = "hello"; string s2 = "world"; string s3 = s1 + " " + s2; // результат: строка "hello world" string s4 = String.Concat(s3, "!!!"); // результат: строка "hello world!!!" Console.WriteLine(s4);

Метод Concat является статическим методом класса String, принимающим в качестве параметров две строки. Также имеются другие версии метода, принимающие другое количество параметров.

Для объединения строк также может использоваться метод Join:

string s5 = "apple"; string s6 = "a day"; string s7 = "keeps"; string s8 = "a doctor"; string s9 = "away"; string[] values = new string[] { s5, s6, s7, s8, s9 }; String s10 = String.Join(" ", values); // результат: строка "apple a day keeps a doctor away"

Метод Join также является статическим. Использованная выше версия метода получает два параметра: строку-разделитель (в данном случае пробел) и массив строк, которые будут соединяться и разделяться разделителем.

**96. Пример переопределения метода toString.**

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public override string ToString()

{

return "Person: " + Name + " " + Age;

}

}

**97. Сравнение строк.**

Статический метод [String.Compare](https://docs.microsoft.com/ru-ru/dotnet/api/system.string.compare) позволяет тщательно сравнивать две строки. Этот метод учитывает язык и региональные параметры. Эту функцию можно использовать для сравнения двух строк или подстрок двух строк. Кроме того, имеются перегруженные методы, которые учитывают или не учитывают регистр и вариативность языка и региональных параметров. В таблице ниже приведены три целочисленных значения, которые может возвращать этот метод.

**98. Как действует == для строк.**

В случае с логическим оператором == (а также !=) сравнивает ссылки на объекты. == сравнивает не по значению, а по ссылке.

**99. Пул строковых литералов.**

Пул строк – множество строк в куче. При загрузке класса все строковые литералы будут занесены в пул строк. Можно занести строку в пул с помощью метода String#intern. Относится к оптимизации компилятора.

**100. Метод intern().**

Извлекает системную ссылку на указанный объект String.

Это метод intern() в классе java.lang.String. Выражение new String("Hello").intern() == new String("Hello").intern()возвращает true, в то время как без использования intern() возвращается false.

**101. Особенности индексации строк.**

Индексация символов в строке, как и элементов массива, начинается с нуля.

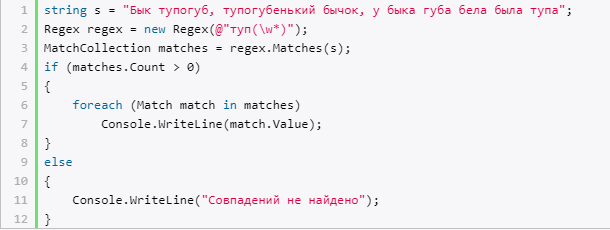
**102. Понятие регулярного выражения**.

Регулярные выражения — это часть небольшой технологической области, невероятно широко используемой в огромном диапазоне программ. Регулярные выражения можно представить себе как мини-язык программирования, имеющий одно специфическое назначение: находить подстроки в больших строковых выражениях.

# **103. Пример использования регулярного выражения.**

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен **System.Text.RegularExpressions**. А центральным классом при работе с регулярными выражениями является класс **Regex**. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:



Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект **Match**. Его свойство Value возвращает найденное совпадение.

**104. Особенности класса StringBuilder.**

Чтобы изменять строку без создания нового объекта, можно использовать класс System.Text.StringBuilder. Например, использование класса StringBuilder может повысить производительность при соединении большого количества строк в цикле.

Касс StringBuilder предлагает ряд методов для операций над строками:

Insert: вставляет подстроку в объект StringBuilder, начиная с определенного индекса

Remove: удаляет определенное количество символов, начиная с определенного индекса

Replace: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку

AppendFormat: добавляет подстроку в конец объекта StringBuilder.

**105. Понятие исключения.**

Как показывает практика программирования, в процессе работы программы могут возникнуть какие-либо неполадки, связанные, например, с некорректными данными. В языке C# такие ситуации обрабатываются с помощью исключений. Говорят, что если операция или метод не может выполниться корректно, то выбрасываются исключения соответствующего типа. Выражение «выбрасываются» надо понимать не буквально, а в том смысле, что при обнаружении ошибки система прерывает выполнение программы и сообщает об этом.

**106. Способы сообщения о возникновении ошибки.**

Инструкция try - отмечает блок кода как объект для обработки ошибок или очистки. После блока try обязательно должен идти либо блок catch, либо блок finally, либо они оба. Блок catch выполняется, когда внутри блока try возникает ошибка. Блок finally выполняется после того, как прекращает выполнять блок try (или, если присутствует, блок catch), независимо от того, выполнился ли он до конца или был прерван ошибкой, что позволяет выполнить так называемый код очистки.

Блок catch - имеет доступ к объекту исключения (Exception), который содержит информацию об ошибке. Блок catch позволяет обработать исключительную ситуацию и как-либо скорректировать ошибку или выбросить новое исключение. Повторное выбрасывание

исключения в блоке catch обычно применяется с целью логирования ошибок или чтобы выбросить новое, более специфическое исключение.

Блок finally - добавляет в программу прогнозируемость, позволяя выполнить определенный код при любых обстоятельствах. Это может быть полезно для выполнения операций очистки, например, закрытия сетевого подключения и т.д.

**107. Преимущества использования исключений.**

Базовым для всех типов исключений является тип **Exception**. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

Прелесть этой методики состоит в том, что она позволяет разработчикам использовать в области обработки ошибок унифицированный подход, который является общим для всех языков, ориентированных на платформу .NET.

Еще одно преимущество механизма исключений .NET состоит в том, что в отличие от запутанных числовых значений, просто обозначающих текущую проблему, они представляют собой объекты, в которых содержится читабельное описание проблемы, а также детальный снимок стека вызовов на момент, когда изначально возникло исключение. Более того, конечному пользователю можно предоставлять справочную ссылку, которая указывает на определенный URL-адрес с описанием деталей ошибки, а также специальные данные, определенные программистом.

**108. Объект-исключение.**

Базовым для всех типов исключений является тип Exception. Этот тип определяет ряд свойств, с помощью которых можно получить информацию об исключении.

* InnerException: хранит информацию об исключении, которое послужило причиной текущего исключения
* Message: хранит сообщение об исключении, текст ошибки
* Source: хранит имя объекта или сборки, которое вызвало исключение
* StackTrace: возвращает строковое представление стека вызывов, которые привели к возникновению исключения
* TargetSite: возвращает метод, в котором и было вызвано исключение

Например, обработаем исключения типа Exception:

static void Main(string[] args) { try { int x = 5; int y = x / 0; Console.WriteLine($"Результат: {y}"); } catch (Exception ex) { Console.WriteLine($"Исключение: {ex.Message}"); Console.WriteLine($"Метод: {ex.TargetSite}"); Console.WriteLine($"Трассировка стека: {ex.StackTrace}"); } Console.Read(); }

Но также есть более специализированные типы исключений, которые предназначены для обработки каких-то определенных видов исключений. Их довольно много, приведём некоторые примеры:

* DivideByZeroException: представляет исключение, которое генерируется при делении на ноль
* ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений
* ArgumentException: генерируется, если в метод для параметра передается некорректное значение
* IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов
* NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

**109. Бросок исключения.**

try { ... }

catch{ throw }

**110. Общий алгоритм обработки исключений.**

Место, в котором может произойти ошибка, должно входить в контролируемый

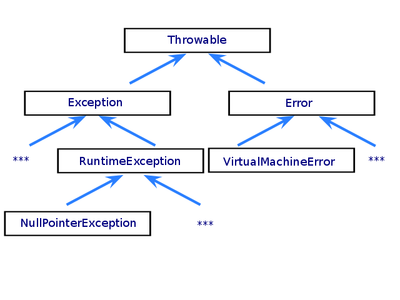
блок — составной оператор, перед которым записано ключевое слово try.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

* Обработка исключения начинается с появления ошибки. Функция, в которой она возникла, генерирует исключение. Для этого используется ключевое слово throw с параметром, определяющим вид исключения. Параметр может быть константой, переменной или объектом и используется для передачи информации об исключении его обработчику.
* Отыскивается соответствующий обработчик исключения и ему передается управление.
* Если обработчик исключения не найден, вызывается стандартная функция terminate, которая вызывает функцию abort, аварийно завершающую текущий процесс. Можно установить собственную функцию завершения процесса.

**111. Классификация исключений.**

Класс Java Throwable описывает все, что может быть брошено как исключение. Наследеники Throwable - Exception и Error - основные типы исключений. Также RuntimeException, унаследованный от Exception, является существенным классом.



**112. Контролируемые и не контролируемые исключения.**

Исключение — это проблема(ошибка) возникающая во время выполнения программы. Исключения могут возникать во многих случаях, например:

1. Пользователь ввел некорректные данные.
2. Файл, к которому обращается программа, не найден.
3. Сетевое соединение с сервером было утеряно во время передачи данных. И т.д.

Все исключения в Java являются объектами. Поэтому они могут порождаться не только автоматически при возникновении исключительной ситуации, но и создаваться самим разработчиком. В Java все исключения делятся на два типа: контролируемые исключения (checked) и неконтролируемые исключения (unchecked), к которым относятся ошибки (Errors) и исключения времени выполнения (RuntimeExceptions, потомок класса Exception).

Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса Exception (но не RuntimeException).

**113. Объявление исключений.**

Чтобы создать объект исключения, программисту требуется вызвать конструктор класса исключений оператором raise (throw).

Блоки кода с определенными ответчиками на исключения называются защищенными блоками, потому что они могут защищать от ошибок, которые иначе могут завершить работу приложения или уничтожить данные. Когда вы объявляете защищенный блок, вы определяете конкретных ответчиков на исключения, которые могут произойти внутри этого блока. Когда ошибка происходит в этом блоке, исключение немедленно "прыгает" в ответчик, определенный вами, и затем покидает блок.

**114. Генерация исключений.**

Метод не способен выполнить свои функции.

throw new System.ArgumentException("Parameter cannot be null", "original");

На основе состояния объекта выполнен неправильный вызов объекта.:

throw new System.InvalidOperationException("Logfile cannot be read-only");

Когда аргумент метода вызывает исключение

System.ArgumentException argEx = new System.ArgumentException("Index is out of range", "index", ex);

throw argEx;

**115. Перехват исключений**.

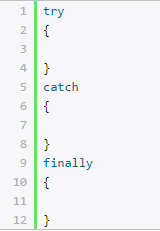
Принимая во внимание, что .NET Framework включает большое количество предопределенных классов исключений, возникает вопрос: как их использовать в коде для перехвата ошибочных условий? Для того чтобы справиться с возможными ошибочными

ситуациями в коде C#, программа обычно делится на блоки трех разных типов:

* Блоки try инкапсулируют код, формирующий часть нормальных действий программы, которые потенциально могут столкнуться с серьезными ошибочными ситуациями.
* Блоки catch инкапсулируют код, который обрабатывает ошибочные ситуации, происходящие в коде блока try. Это также удобное место для протоколирования ошибок.
* Блоки finally инкапсулируют код, очищающий любые ресурсы или выполняющий другие действия, которые обычно нужно выполнить в конце блоков try или catch. Важно понимать, что этот блок выполняется независимо от того, сгенерированo исключение или нет.

# **116. Формат инструкции try-catch-finnaly.**

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**.

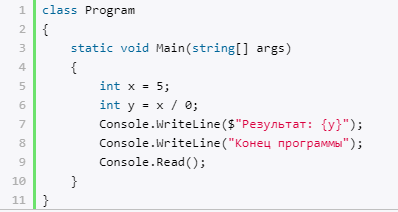


При использовании блока **try...catch..finally** вначале выполняются все инструкции в блоке **try**. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally**. И затем конструкция try..catch..finally завершает свою работу.

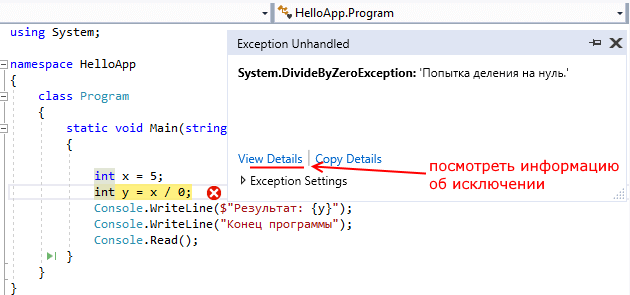
Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок **catch**, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

Рассмотрим следующий пример:



В данном случае происходит деление числа на 0, что приведет к генерации исключения. И при запуске приложения в режиме отладки мы увидим в Visual Studio окошко, которое информирует об исключении:



В этом окошке мы видим, что возникло исключение, которое представляет тип **System.DivideByZeroException**, то есть попытка деления на ноль. С помощью пункта **View Details** можно посомтреть более детальную информацию об исключении.

И в этом случае единственное, что нам остается, это завершить выполнение программы.

Чтобы избежать подобного аварийного завершения программы, следует использовать для обработки исключений конструкциюtry...catch...finally. Так, перепишем пример следующим образом:

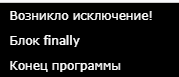


В данном случае у нас опять же возникнет исключение в блоке try, так как мы пытаемся разделить на ноль. И дойдя до строки



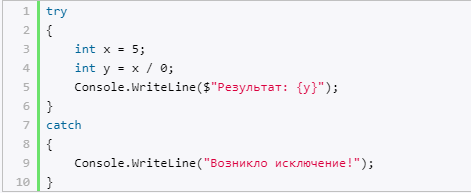
выполнение программы остановится. CLR найдет блок **catch** и передаст управление этому блоку.

После блока catch будет выполняться блок finally.

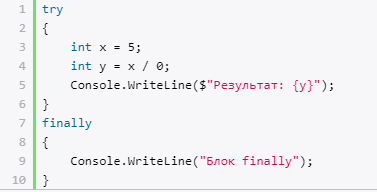


Таким образом, рограмма по-прежнему не будет выполнять деление на ноль и соответственно не будет выводить результат этого деления, но теперь она не будет аварийно завершаться, а исключение будет обрабатываться в блоке catch.

Следует отметить, что в этой конструкции обязателен блок **try**. При наличии блока catch мы можем опустить блок finally:



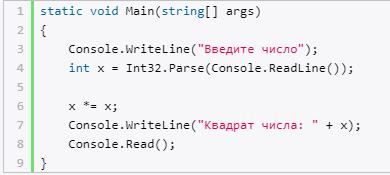
И, наоборот, при наличии блока finally мы можем опустить блок catch и не обрабатывать исключение:



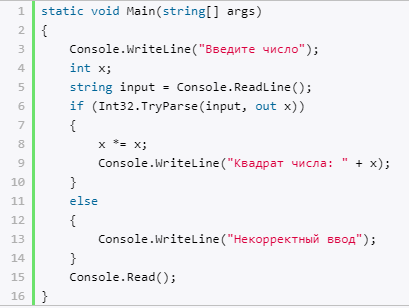
Однако, хотя с точки зрения синтаксиса C# такая конструкция вполне корректна, тем не менее, поскольку CLR не сможет найти нужный блок catch, то исключение не будет обработано, и программа аварийно завершится.

## Обработка исключений и условные конструкции

Ряд исключительных ситуаций может быть предвиден разработчиком. Например, пусть программа предусматривает ввод числа и вывод его квадрата:



Если пользователь введет не число, а строку, какие-то другие символы, то программа выпадет в ошибку. С одной стороны, здесь как раз та ситуация, когда можно применить блок try..catch, чтобы обработать возможную ошибку. Однако гораздо оптимальнее было бы проверить допустимость преобразования:



Метод Int32.TryParse() возвращает true, если преобразование можно осуществить, и false - если нельзя. При допустимости преобразования переменная x будет содержать введенное число. Так, не используя try...catch можно обработать возможную исключительную ситуацию.

С точки зрения производительности использование блоков try..catch более накладно, чем применение условных конструкций. Поэтому по возможности вместо try..catch лучше использовать условные конструкции на проверку исключительных ситуаций.

**117. Пример обработки исключения.**

Для этого в C# предназначена конструкция try...catch...finally.

try { } catch { } finally { } Например: class Program { static void Main(string[] args) { int x = 5; int y = x / 0; Console.WriteLine($"Результат: {y}"); Console.WriteLine("Конец программы"); Console.Read(); } } В данном случае происходит деление числа на 0, что приведет к генерации исключения Чтобы избежать аварийного завершения программы, следует использовать для обработки исключений конструкциюtry...catch...finally

class Program

{

static void Main(string[] args)

{

try

{

int x = 5;

int y = x / 0;

Console.WriteLine($"Результат: {y}");

}

catch

{

Console.WriteLine("Возникло исключение!");

}

finally

{

Console.WriteLine("Блок finally");

}

Console.WriteLine("Конец программы");

Console.Read();

}

}

В данном случае у нас опять же возникнет исключение в блоке try, так как мы пытаемся разделить на ноль

выполнение программы остановится. CLR найдет блок catch и передаст управление этому блоку.

После блока catch будет выполняться блок finally.

Возникло исключение!

Блок finally

Конец программы

**118. Основные контролируемые исключения в C#.**

* DivideByZeroException: представляет исключение, которое генерируется при делении на ноль
* ArgumentOutOfRangeException: генерируется, если значение аргумента находится вне диапазона допустимых значений
* ArgumentException: генерируется, если в метод для параметра передается некорректное значение
* IndexOutOfRangeException: генерируется, если индекс элемента массива или коллекции находится вне диапазона допустимых значений
* InvalidCastException: генерируется при попытке произвести недопустимые преобразования типов
* NullReferenceException: генерируется при попытке обращения к объекту, который равен null (то есть по сути неопределен)

**119. Основные неконтролируемые исключения в C#.**

System.ArgumentException — выбрасывается при вызове функции с неправильным аргументом.

System.ArgumentNullException — производный от ArgumentException класс, выбрасывается если один из аргументов функции неожиданно равен null.

System.ArgumentOutOfRangeException — производный от ArgumentException класс, выбрасывается когда аргумент функции имеет слишком большое или слишком маленькое значение для данного типа (обычно касается числовых типов). Например, такое исключение будет выброшено если попытаться передать отрицательное число в функцию, которая ожидает только положительные числа.

System.InvalidOperationException — выбрасывается когда состояние объекта является неподходящим для нормального выполнения метода, например, при попытке прочесть не открытый файл.

System.NotSupportedException — выбрасывается, когда запрошенный функционал не поддерживается, например, если попытаться вызвать метод Add для коллекции доступной только для чтения (свойство коллекции IsReadOnly возвращает true).

System.NotImplementedException — выбрасывается, когда запрошенный функционал еще не реализован.

System.ObjectDisposedException — выбрасывается при попытке вызвать метод объекта, который уже был уничтожен (disposed).