1.

- Разработка через тестирование (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

- Каскадная модель (англ. waterfall model, иногда переводят как модель «Водопад») — модель процесса разработки программного обеспечения, в которой процесс разработки выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки.

- Итеративный подход (англ. iteration - «повторение») в разработке программного обеспечения — это выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы. Проект при этом подходе в каждой фазе развития проходит повторяющийся цикл PDCA: Планирование — Реализация — Проверка — Оценка (англ. plan-do-check-act cycle).

2. Объе́ктно-ориенти́рованное программи́рование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Объектно-ориентированный язык программирования (ОО-язык) — язык, построенный на принципах объектно-ориентированного программирования.

В основе концепции объектно-ориентированного программирования лежит понятие объекта — некой сущности, которая объединяет в себе поля (данные) и методы (выполняемые объектом действия).

Примеры языков: C++, C#, Java, JavaScript, Python, Swift, PHP, Objective-C.

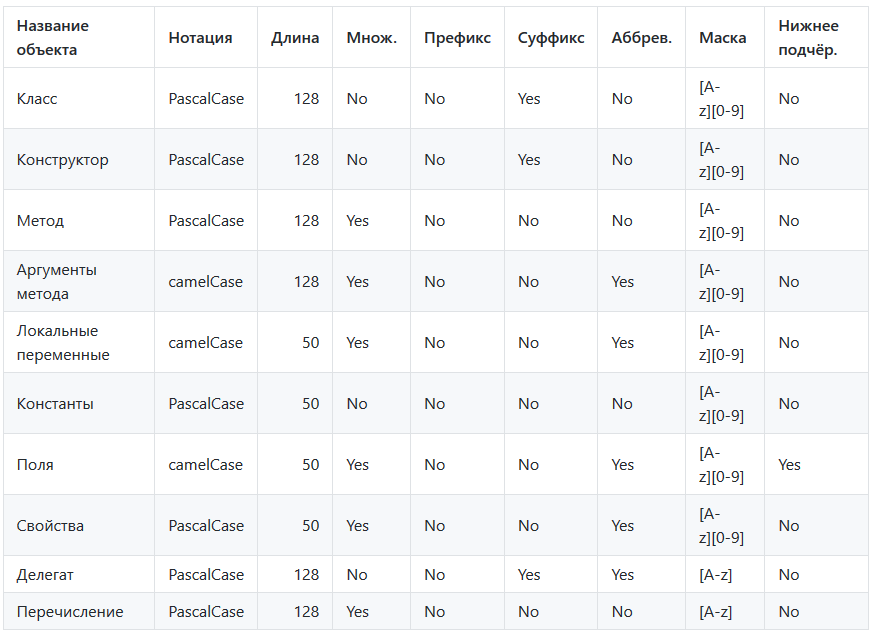
Процеду́рное программи́рование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка.

Процедурный язык программирования предоставляет возможность программисту определять каждый шаг в процессе решения задачи. Особенность таких языков программирования состоит в том, что задачи разбиваются на шаги и решаются шаг за шагом. Используя процедурный язык, программист определяет языковые конструкции для выполнения последовательности алгоритмических шагов.

Примеры языков: C, Go, Basic, Pascal.

Среды разработки: IntelliJ IDEA, NetBeans, Eclipse, Qt Creator, Geany, Embarcadero RAD Studio, Code::Blocks, Xcode, Microsoft Visual Studio.

3. Станда́рт оформле́ния ко́да (станда́рт коди́рования, стиль программи́рования) (англ. coding standards, coding convention или programming style) — набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.



4. Абстрактный класс в объектно-ориентированном программировании — базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (и не содержать) абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание.

abstract class MyClass {…}

5. Перечисляемый тип (сокращённо перечисле́ние, англ. enumeration, enumerated type) — в программировании тип данных, чьё множество значений представляет собой ограниченный список идентификаторов.

enum Cardsuit { Clubs, Diamonds, Spades, Hearts }

6. Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

Классы ООП состоят из элементов различных типов:

Поля данных: хранят состояние класса с помощью переменных и структур.

Методы: подпрограммы для манипулирования указанными данными.

Некоторые языки допускают третий тип - свойства. Это что-то среднее между первыми двумя.

7.

Конструктор класса (от англ. constructor) — специальный блок инструкций, вызываемый при создании объекта. Записывается как метод, только с названием класса и без возвращаемого типа.

Дестру́ктор — специальный метод класса, служащий для деинициализации объекта (например освобождения памяти). Записывается так, как конструктор, только перед названием ставится тильда (~).

8. Как правило, после создания объекта, его нужно «привести» в некое начальное состояние, т.е. инициализировать.

- Первый случай — это простое создание объекта с вызовом конструктора без аргументов

SomeClass tmpObj = new SomeClass();

- Следующий вариант — это вызов конструктора с аргументами.

Person somePerson = new Person("Иван", 26);

- Третий вариант инициализации приходится использовать тогда, когда «базовый» вариант не походит (как и во втором случае), но в классе нет конструктора, удовлетворяющего всем потребностям.

Person somePerson = new Person("Иван", 26);

somePerson.Height = 190;

- Можно задать значения открытым свойствам сразу после создания объекта.

Person somePerson = new Person() { Name = "Иван", Age = 26, Height = 190 };

9. Классы могут быть разделяемыми (частичными). То есть мы можем иметь несколько файлов с определением одного и того же класса, и при компиляции все эти определения будут скомпилированы в одно. Для этого используется ключевое слово partial.

Такие классы могут иметь разделяемые методы. Причём определение такого метода может находиться в одном разделяемом классе, а реализация в другом.

10. Структура - композитный тип данных, инкапсулирующий без сокрытия набор значений различных типов. Структура является типом значения, то есть не относится к ссылочным типам данных, как например класс, следовательно, она хранится в стеке. Структуры не поддерживают наследование.

11. Пространства имен (namespace) — это способ, благодаря которому .NET избегает конфликтов имен между классами. Они предназначены для того, чтобы исключить ситуации, когда вы определяете класс, представляющий заказчика, называете его Customer, а после этого кто-то другой делает то же самое (подобный сценарий достаточно распространен).

Пространство имен — это не более чем группа типов данных, но дающая тот эффект, что имена всех типов данных в пределах пространства имен автоматически снабжаются префиксом - названием пространства имен. Пространства имен можно вкладывать друг в друга. Например, большинство базовых классов .NET общего назначения находятся в пространстве имен System. Базовый класс Array относится к этому пространству, поэтому его полное имя — System.Array.

12. Наследование (англ. inheritance) — концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения.

Наследование является механизмом повторного использования кода и способствует независимому расширению программного обеспечения через открытые классы и интерфейсы. Установка отношения наследования между классами порождает иерархию классов.

13. Инкапсуляция (англ. encapsulation, от лат. in capsula) — упаковка данных и функций в единый компонент.

В общем случае в разных языках программирования термин «инкапсуляция» относится к одной или обеим одновременно следующим нотациям:

- механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;

- языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

Пренебрегая формализмом и способствуя интуитивному восприятию, инкапсуляцию можно определить с помощью латинского in capsula — размещение в оболочке, изоляция, закрытие чего-либо инородного с целью исключения влияния на окружающее, обеспечение доступности главного, выделение основного содержания путём помещения всего мешающего, второстепенного в некую условную капсулу (чёрный ящик). Однако данное определение является лишь приближением.

В C# часто для инкапсуляции используются свойства.

14. Полиморфизм — способность функции обрабатывать разные типа данных. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

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

- внешняя общность проявляется как одинаковый набор методов с одинаковыми именами и сигнатурами (именем методов и типами аргументов, и их количеством);

- внутренняя общность — одинаковая функциональность методов. Её можно описать интуитивно или выразить в виде строгих законов, правил, которым должны подчиняться методы. Возможность приписывать разную функциональность одному методу (функции, операции) называется перегрузкой метода (перегрузкой функций, перегрузкой операций).

15. Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

16.

17. Работая с системой Visual Studio, пользователь открывает решение. При повторном редактировании специальных файлов создается временное решение, которое можно уничтожить по окончании работы. Однако решение позволяет управлять текущими файлами, поэтому в большинстве случаев его сохранение означает, что пользователь может вернуться к тому, что он делал накануне, и вновь открыть файлы, с которыми он работал.

Решение можно интерпретировать как контейнер связанных между собой проектов. Проекты внутри решения не обязательно должны быть написаны на одном и том же языке программирования или иметь одинаковый тип.

Наиболее распространенным способом структурирования приложений в среде Visual Studio является одно отдельное решение, содержащее много проектов. Каждый проект можно создать из набора исходных файлов и папок.

18.

19. Отладка, точки останова, вывод значений переменных.

20. .NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения Common Language Runtime (CLR), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

21. Когда мы создаем приложение в результате компиляции в Visual Studio или в консоли, результатом этой работы является файл exe или dll (в зависимости от выбранных настроек), который называется сборкой приложения. Сборка является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

Сборки кристаллизуют всю библиотеку классов .NET - при написании кода и создании сборки своего приложения мы используем пространства имен, которые размещены в других сборках .NET.

Сборки имеют следующие составляющие:

- Манифест, который содержит метаданные сборки

- Метаданные типов. Используя эти метаданные, сборка определяет местоположение типов в файле приложения, а также места размещения их в памяти

- Собственно код приложения на языке MSIL, в который компилируется код C#

- Ресурсы

Все эти компоненты могут находиться в одном файле, и тогда сборка представляет один единственный файл в формате exe или dll.

22. Ядром платформы .NET Framework является общеязыковая исполняющая среда (Common Language Runtime) или сокращенно CLR.

На компьютере храниться в виде файла динамической библиотеки mscoree.dll, в проекте данная сборка подключается автоматически.

Какие задачи решает CLR?

- Загрузка пользовательских сборок

- Управление памятью

- Работа с потоками

- Безопасность выполнения кода

- Очистка памяти (сборщик мусора)

- и многое другое

CLR компилирует код приложения на языке CIL (реализация компиляции которого компанией Microsoft называется MSIL) во время его исполнения, а также предоставляет MSIL-программам (а следовательно, и программам, написанным на языках высокого уровня, поддерживающих .NET Framework) доступ к библиотеке классов .NET Framework, или так называемой .NET FCL (англ. Framework Class Library).

23.

24. Генератор документации — программа или пакет программ, позволяющая получать документацию, предназначенную для программистов (документация на API) и/или для конечных пользователей системы, по особому образу комментированному исходному коду и, в некоторых случаях, по исполняемым модулям (полученным на выходе компилятора).

25.

Типы значений

Простые типы: sbyte, short, int, long – со знаком целочисленные; byte, ushort, uint, ulong – без знака; сhar – символы Юникода; float, double – вещественные; decimal – с повышенной точностью; bool – логическое.

Перечисление: enum.

Структуры: struct.

Ссылочные типы

Типы классов: object – исходный базовый класс; string – строки Юникода; class C {…} – пользовательский.

Типы интерфейса: interface I {...}.

Типы массивов: int[] или int[, ].

Типы делегатов: delegate int D(...).

Неявно типизированная переменная – var. Она получает определённый тип при присваивании значения, дальше этот тип не изменяется.

Тип dynamic – почти как var, но dynamic переменная не строго типизирована на протяжении всего времени её существования.

26. Локальная, Глобальная, Пакет или пространство имён, также private, protected, public.

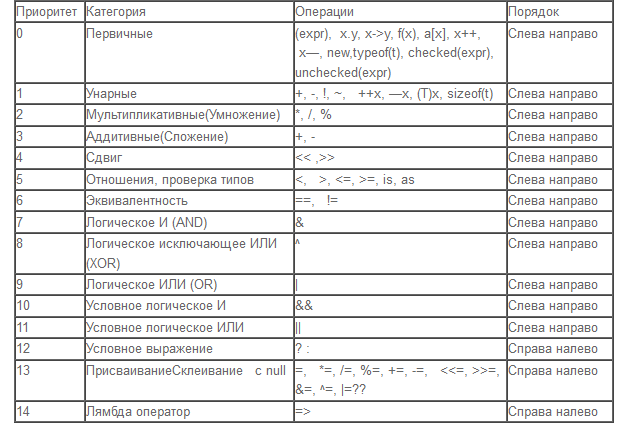
27.

- Неявное преобразование (implicit conversion), для корректности проверяются типы (не значения).

- Явное преобразование (explicit conversion), если сужающее, то можно использовать проверку переполнения с помощью ключевого слова checked, если переполнение произошло, то выбрасывает OverflowException.

28. Константа (const) проинициализирована при объявлении и её значение не может быть изменено.

Только для чтения (readonly) может быть проинициализирована при объявлении либо в конструкторе класса, изменять после инициализации нельзя.

29. 

30. Оператор ветвления

if (условие)

оператор (операторы)

else

оператор (операторы)

31. Оператор выбора

switch(выражение) {

case константа1:

последовательность операторов

break;

case константа2:

последовательность операторов

break;

case константаЗ:

последовательность операторов

break;

...

default:

последовательность операторов

break;

}

32.

for (int i = 0; i < 9; i++)

{

Console.WriteLine($"Квадрат числа {i} равен {i\*i}");

}

int i = 6;

while (i > 0)

{

Console.WriteLine(i);

i--;

}

int i = 6;

do

{

Console.WriteLine(i);

i--;

}

while (i > 0);

foreach (тип имя\_переменной\_цикла in коллекция)

оператор;

break – для выхода из цикла, continue – для пропуска итерации.

33.

int[] arr = new int[133];

int[] arr = new int[] {12,23,34,45, 56};

int[] arr = {0,4,8,32};

int[] arr = new int[3] {-5,2,3};

var varArr = new[] { "123", null, "exe" };

34. Обычный массив только с var

var varArr = new[] { "file", null, "apricot" };

35.

Каждый создаваемый массив получает большую часть функциональности от класса System.Array. Общие члены этого класса позволяют работать с массивом с использованием полноценной объектной модели. Таким образом, методы и свойства, определенные в классе Array, можно использовать с любым массивом C#. Давайте разберем некоторые полезные особенности класса Array.

Класс Array является абстрактным, поэтому создать массив с использованием какого-либо конструктора нельзя. Однако вместо применения синтаксиса C# для создания экземпляров массивов также возможно создавать их с помощью статического метода CreateInstance(). Это исключительно удобно, когда заранее неизвестен тип элементов массива, поскольку тип можно передать методу CreateInstance() в параметре как объект Type.

// Создаем массив типа string, длиной 5

Array myArr = Array.CreateInstance(typeof(string),5);

// Инициализируем первые два поля массива

myArr.SetValue("Name",0);

myArr.SetValue("Age",1);

// Считываем данные из массива

string s = (string)myArr.GetValue(1);

36. break, continue

37. Символы и строки

38. В языке C# строковые значения представляет тип string, а вся функциональность работы с данным типом сосредоточена в классе System.String. Собственно, string является псевдонимом для класса System.String. Объекты этого класса представляют текст как последовательность символов Unicode. Максимальный размер объекта String может составлять в памяти 2 ГБ, или около 1 миллиарда символов.

string s1 = "hello";

string s2 = null;

string s3 = new String('a', 6); // результатом будет строка "aaaaaa"

string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'});

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

39. Динамическая строка StringBuilder. Операции не создают новые строки.

Метод Append - этот метод добавляет к строке подстроку.

Кроме метода Append класс StringBuilder предлагает еще 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

40.

Регуля́рные выраже́ния (англ. regular expressions) — формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов (символов-джокеров, англ. wildcard characters). Для поиска используется строка-образец (англ. pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.

Некоторые элементы синтаксиса регулярных выражений:

^: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")

$: конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)

.: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")

\*: предыдущий символ повторяется 0 и более раз

+: предыдущий символ повторяется 1 и более раз

?: предыдущий символ повторяется 0 или 1 раз

\s: соответствует любому пробельному символу

\S: соответствует любому символу, не являющемуся пробелом

\w: соответствует любому алфавитно-цифровому символу

\W: соответствует любому не алфавитно-цифровому символу

\d: соответствует любой десятичной цифре

\D : соответствует любому символу, не являющемуся десятичной цифрой

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex.

41.

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else

{

Console.WriteLine("Совпадений не найдено");

}

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

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

Проверка на соответствие строки формату IsMatch.

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой.

42.

[модификаторы] тип\_возвращаемого\_значения название\_метода ([параметры])

{

// тело метода

}

43. Перегруженные методы должны отличаться по:

- Количеству параметров

- Типу параметров

- Порядку параметров

- Модификаторам параметров

Add(int, int)

Add(int, int, int)

Add(int, int, int, int)

Add(double, double)

Add(int, double)

Add(double, int)

Add(ref int, ref int)

44. params

static void Addition( int x, string mes, params int[] integers)

{}

45.

// Аргументы b и с указывать при вызове необязательно

static int mySum(int a, int b = 5, int c = 10)

{

return a + b + c;

}

// Использование именованных аргументов

// при вызове метода

int sum1 = mySum(a: 3, b: 10);

46. Выше мы использовали входные параметры. Но параметры могут быть также выходными. Чтобы сделать параметр выходным, перед ним ставится модификатор out.

static void Sum(int x, int y, out int a)

{

a = x + y;

}

int z;

Sum(x, 15, out z);

47.

Типы значений:

Целочисленные типы (byte, sbyte, char, short, ushort, int, uint, long, ulong)

Типы с плавающей запятой (float, double)

Тип decimal

Тип bool

Перечисления enum

Структуры (struct)

Ссылочные типы:

Тип object

Тип string

Классы (class)

Интерфейсы (interface)

Делегаты (delegate)

С помощью ref можно передать переменную по ссылке, работает, как и out, но с ref всегда нужно использовать инициализированные переменные, иначе будет ошибка.

48.

- Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе).

- Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

49.

Значение null по умолчанию могут принимать только объекты ссылочных типов. Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных - какое-то определенное значение или же null. Для этого надо использовать знак вопроса ? после типа значений.

int? z1 = 5;

bool? enabled1 = null;

Double? d1 = 3.3;

Но фактически запись ? является упрощенной формой использования структуры System.Nullable<T>. Параметр T в угловых скобках представляет универсальный параметр, вместо которого в конкретной задача уже подставляется конкретный тип данных.

Nullable<int> z2 = 5;

Nullable<bool> enabled2 = null;

Nullable<System.Double> d2 = 3.3;

Если использовать ссылочный тип (класс), то будет ошибка.

50.

Модульное тестирование, или юнит-тестирование (англ. unit testing) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Рефа́кторинг (англ. refactoring), или перепроектирование кода, переработка кода, равносильное преобразование алгоритмов — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы. В основе рефакторинга лежит последовательность небольших эквивалентных (то есть сохраняющих поведение) преобразований. Поскольку каждое преобразование маленькое, программисту легче проследить за его правильностью, и в то же время вся последовательность может привести к существенной перестройке программы и улучшению её согласованности и чёткости.

51.

Класс Assert из пространства имен Microsoft.VisualStudio.TestTools.UnitTesting с помощью своих статических методов позволяет верифицировать результат выполнения некоторого действия. Ранее уже было рассмотрено несколько методов, в частности, метод Assert.IsNotNull(), проверяющий, не равен ли некоторый объект значению null. Кроме того, при тестировании нам доступен еще ряд методов:

AreEqual(object expected, object actual): проверяет, равны ли оба объекта. Имеет различные перегруженные версии, позволяющие сравнивать различные типы объектов

AreEqual<T>(T expected, T actual): обобщенная версия предыдущего метода. Например, Assert.AreEqual<string>("Index", result.MasterName)

AreNotEqual(object expected, object actual): проверяет, не равны ли оба объекта. Тест проходит успешно, если объекты не равны

AreNotEqual<T>(T expected, T actual): обобщенная версия предыдущего метода

AreSame(object expected, object actual): проверяет, указывают ли оба объекта на один и тот же объект в памяти

AreNotSame(object expected, object actual): проверяет, указывают ли оба объекта на разные объекты в памяти. Если они указывают на один и тот же объект, то тест заканчивается неудачно

Equals(object objA, object objB): проверяет на равенство оба объекта

IsFalse(bool condition): проверяет, равно ли условие condition значению false

IsTrue(bool condition): проверяет, равно ли условие condition значению true

IsNull(object value): проверяет, имеет ли объект value значение null

IsInstanceOfType(object value, Type expectedType): проверяет, представляет ли объект value тип expectedType

52.

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.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором private. Классы, объявленные без модификатора, по умолчанию имеют доступ internal.

53.

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют свойства. Они обеспечивают простой доступ к полям класса, узнать их значение или выполнить их установку.

[модификатор\_доступа] возвращаемый\_тип произвольное\_название

{

// код свойства

}

Автоматические свойства: public string Name { get; set; }

54. Индексаторы позволяют индексировать объекты и обращаться к данным по индексу. Фактически с помощью индексаторов мы можем работать с объектами как с массивами. По форме они напоминают свойства со стандартными блоками get и set, которые возвращают и присваивают значение.

возвращаемый\_тип this [Тип параметр1, ...]

{

get { ... }

set { ... }

}

55. Статические (static) поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса. Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.

56. Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров

- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово this для ссылки на текущий объект класса и можно обращаться только к статическим членам класса

- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются)

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз.

Статические классы объявляются с модификатором static и могут содержать только статические поля, свойства и методы.

57. Методы расширения (extension methods) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса. Эта функциональность бывает особенно полезна, когда нам хочется добавить в некоторый тип новый метод, но сам тип (класс или структуру) мы изменить не можем, поскольку у нас нет доступа к исходному коду. Либо если мы не можем использовать стандартный механизм наследования, например, если классы определенны с модификатором sealed.

Например, нам надо добавить для типа string новый метод:

public static class StringExtension

{

public static int WordCount(this string str, char c)

{

int counter = 0;

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

{

if (str[i] == c)

counter++;

}

return counter;

}

}

Вызываем метод таким способом:

int i = s.WordCount(c);

Перегрузка операций (операторов):

public static возвращаемый\_тип operator оператор(параметры)

{ }

Можно перегрузить следующие операторы:

унарные операторы +, -, !, ~, ++, --

бинарные операторы +, -, \*, /, %

операции сравнения ==, !=, <, >, <=, >=

логические операторы &&, ||

операторы присваивания +=, -=, \*=, /=, %=

58.

Перегрузка преобразования неявная|явная:

public static implicit|explicit operator Тип\_в\_который\_надо\_преобразовать(исходный\_тип param)

{

// логика преобразования

}

59. 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.

В целом интерфейсы могут определять следующие сущности:

- Методы

- Свойства

- Индексаторы

- События

Однако интерфейсы не могут определять статические члены, переменные, константы.

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.

Неявная имплементация (реализация) всегда является открытой (public), поэтому к методам и свойствам можно обращаться напрямую.

var imp = new ImplicitTest();

imp.DoTest();

Явная имплементация всегда закрыта (private).

Чтобы получить доступ к имплементации необходимо кастовать инстанцию класса к интерфейсу (upcast to interface).

var exp = new ExplicitTest();

((ITest)exp).DoTest();

60.

Те методы и свойства, которые мы хотим сделать доступными для переопределения, в базовом классе помечается модификатором virtual. Такие методы и свойства называют виртуальными.

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

Фактически сокрытие представляет определение в классе-наследнике метода или свойства, которые соответствует по имени и набору параметров методу или свойству базового класса. Для сокрытия членов класса применяется ключевое слово new.

Если метод в базовом классе не является виртуальным, мы не можем его переопределить, но, допустим, нас не устраивает его реализация для производного класса, поэтому мы можем воспользоваться сокрытием, чтобы определить нужный нам функционал.

61. Также можно запретить переопределение методов и свойств. В этом случае их надо объявлять с модификатором sealed.

При создании методов с модификатором sealed надо учитывать, что sealed применяется в паре с override, то есть только в переопределяемых методах.

Пример:

public override sealed void Display()

{

Console.WriteLine($"{FirstName} {LastName} работает в {Company}");

}

Также можно использовать модификатор sealed для классов.

62. Класс Person имеет конструктор, который устанавливает свойство Name. Поскольку класс Employee наследует и устанавливает то же свойство Name, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса Person. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова base мы можем обратиться к базовому классу. В нашем случае в конструкторе класса Employee нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса Person, с помощью выражения base(name).

class Person

{

public string Name { get; set; }

public Person(string name)

{

Name = name;

}

public void Display()

{

Console.WriteLine(Name);

}

}

class Employee : Person

{

public string Company { get; set; }

public Employee(string name, string company)

: base(name)

{

Company = company;

}

}

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

public override void GetInfo()

{

// Вызов метода GetInfo из базового класса

base.GetInfo();

Console.WriteLine("Employee ID: {0}", id);

}

63.

public interface A

{

int Sum();

}

// Унаследованный интерфейс

public interface B : A

{

int Del();

}

class MyOperation : B

{

int x = 10, y = 5;

public int Sum()

{

return x + y;

}

public int Del()

{

return x / y;

}

}