**Теория**

**Система контроля версий** - программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости, возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение и многое другое.

Системы кв бывают таких типов как:

* Локальные(Revision control System , Rcs)
* Централизованные(Centraliazed version control system)
* Распределенные(Distributed version control system, DVCS)

**Локальные** :

Хранит записи о всех изменениях в файлах.

**Централизованные:**

Взаимодействовать с другими разработчиками.

**Распределенные:**

Клиенты могут полностью копировать репозиторий.

**Git** – это распределённая система управления версиями файлов.

Git предоставляет гибкие и простые в использовании инструменты для ведения истории проекта.

Особенностью Git является то, что работа над версиями проекта может происходить не в хронологическом порядке. Разработка может вестись в нескольких параллельных ветвях, которые могут сливаться и разделяться в любой момент проектирования.

**Объекты Git:**

**1. Деревья**– Они решают проблему хранения имён файлов, а также позволяют хранить группы файлов вместе.

**2. Commit**– описывает снимок в истории

**3. Ветвь** – символьная ссылка (Symbolic link), которая указывает на последний в хронологии commit определенной ветви.

**4. Тег**– тип данных, который в отличие от ветвей неизменно ссылается на один и тот же объект типа tree, commit или tag.

**NET Framework** — программная платформа.

Состоит из **2х** частей:

* **FCL**
* **CLR**

**Base Class Librar**y, или так называемая **.NET FCL** (англ. Framework Class Library), сокращённо **BCL** — стандартная библиотека классов платформы «.NET Framework».

**Common Language Runtime** (англ. **CLR** — общеязыковая исполняющая среда) — исполняющая среда для байт-кода CIL (MSIL), в которой компилируются программы, написанные на .NET-совместимых языках программирования

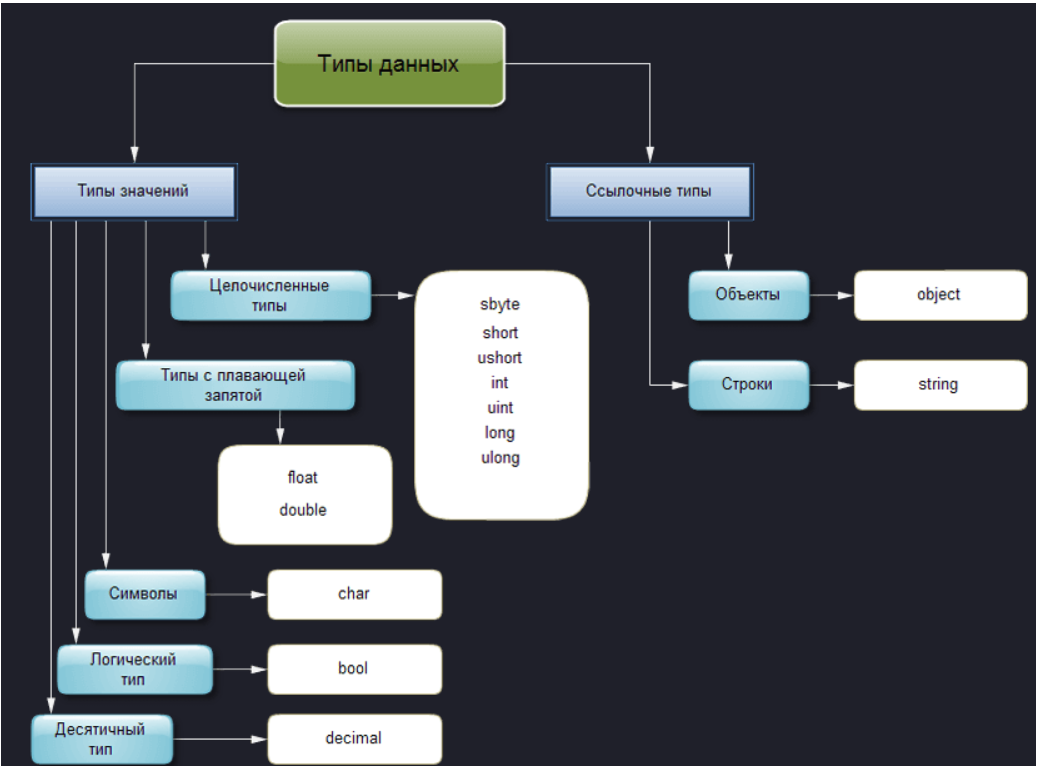
**Portable Executable** (PE, «переносимый исполняемый»). Формат PE представляет собой структуру данных, содержащую всю информацию, необходимую PE-загрузчику для отображения файла в память. Исполняемый код включает в себя ссылки для связывания динамически загружаемых библиотек, таблицы экспорта и импорта API функций, данные для управления ресурсами и данные локальной памяти потока

# **Сборка** является базовой структурной единицей в .NET, на уровне которой проходит контроль версий, развертывание и конфигурация приложения.

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

**Типы данных С#:** C# имеются две общие категории встроенных типов данных: типы значений и ссылочные типы. Они отличаются по содержимому переменной. Концептуально разница между ними состоит в том, что тип значения (value type) хранит данные непосредственно, в то время как ссылочный тип (reference type) хранит ссылку на значение.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как стек, а ссылочные типы — в области, называемой управляемой кучей.



Так же к ссылочным типам относят: Массивы, классы, интерфейсы, делегаты, а к типам значений можно отнести ещё: Перечисление enum, структуры (struct).

**CTS**(общая система типов) представляет собой формальную спецификацию, в которой описано то, как должны быть определены типы для того, чтобы они могли обслуживаться в CLR-среде.

Object – базовый тип данных, который является прародителем любого типа данных С#, соответственно этот тип может хранить в себе абсолютно любой другой тип данных(все типы данных наследуются от object).

object a = 22;

object b = 3.14;

object c = "hello code";

Dynamic – специфический тип данных, который позволяет опустить проверку типов во время компиляции. Кроме того, объекты, объявленные как dynamic, могут в течение работы программы менять свой тип.

dynamic obj = 3;            // здесь obj - целочисленное int

Console.WriteLine(obj);     // 3

obj = "Hello world";        // obj - строка

Console.WriteLine(obj);     // Hello world

Int32 demo = 1; //0

dynamic value;

value = (demo == 0) ?(dynamic)5 : (dynamic)"A";

value = value + value;

Упаковка и распаковка - механизм перемещения данных из области стека в кучу - и наоборот.

Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется упаковкой.

Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется распаковкой.

Int i = 123;

Object o = i;

Int j = (int) j;

Определение типа переменной, обозначенной ключевым словом var, происходит на этапе компиляции, а фактический тип переменной обозначенной ключевым словом **dynamic** определяется в процессе выполнения приложения.

Неявно типизированная переменная – это переменная, которая позволяет компилятору возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Неявно типизированная переменная объявляется с помощью ключевого слова var и должна быть непременно инициализирована.

var i = 12; // переменная i инициализируется целочисленным литералом

var d = 12.3; // переменная d инициализируется литералом с плавающей точкой,

// имеющему тип double

var f = 0.34F; // переменная f теперь имеет тип float

Единственное отличие неявно типизированной переменной от обычной, явно типизированной переменной, — в способе определения ее типа. Как только этот тип будет определен, он закрепляется за переменной до конца ее существования.

В отличие от ссылочных типов переменным/параметрам значимых типов нельзя напрямую присвоить значение **null**.

Чтобы присвоения переменной или параметру значимого типа значения **null**, эти переменная/параметр значимого типа должны представлять тип **nullable**. Для этого после названия типа указывается знак вопроса **?**

int? val = null;

Console.WriteLine(val);

Представление строкового литерала: Console.WriteLine("hello");

Console.WriteLine("фыва");

Console.WriteLine("hello word");

**Null-объединение**  
Оператор **??** называется оператором **null-объединения**. Он применяется для установки значений по умолчанию для типов значений и ссылочных типов, которые допускают значение null. Оператор ?? возвращает левый операнд, если этот операнд не равен null. Иначе возвращается правый операнд. При этом левый операнд должен принимать null.  
При проверке объектов на равенство следует учитывать, что они равны не только, когда они имеют ненулевые значения, которые совпадают, но и когда оба объекта равны null

int? x = null;

int y = x ?? 100;  // равно 100, так как x равен null

int? z = 200;

int t = z ?? 44; // равно 200, так как z не равен null

Над строками определены следующие операции:

- присваивание (=);

- две операции проверки эквивалентности (= =) и (!=);

- конкатенация или сцепление строк (+);

- взятие индекса ([]).

**Основные методы строк**

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, начиная с определенного индекса в массив
* **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

К строкам можно применять операции сравнения. Оператор == возвращает true, если все символы обеих строк равны. Операция **!=** возвращает true, если две строки не совпадают. Так же строки можно сравнивать с помощью метода Compare();

Строковый литерал представляет собой набор символов, заключенных в двойные кавычки. Например следующий фрагмент кода:

"This is text"

Также можно указать буквальный строковый литерал. Такой литерал начинается с символа @, после которого следует строка в кавычках.

Объекты типа StringBuilder отличаются от объектов типа String тем, что они могут быть изменены. По сути они представляют из себя массивы переменной длины, содержащие последовательность символов. Длина и содержание последовательности могут быть изменены вызовом соответствующих методов.

Отличие пустой и null строки заключается в следующем: **string s = “ ”, это пустое значение(**Для пустых строк можно вызывать методы.**), т.е ссылка s указывает на объект в куче, а string s = null, показывает, что это объект не проинициализирован и ни на что в куче не указывает, т.е пустая ссылка.**

**Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем.**

Многомерный массив — это массив, элементы которого являются массивами.

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.

Многомерный массив с разными типами, пример:

object[][] obj = new object[3][];

obj [0] = new bool[] { true,false };

obj [1] = new string[] { "test","test2","test3"};

obj [2] = new int[] { 1,2,3 };

Пример одномерного массива:int[] mas = new int[10]{1,2,3,4,5,6,7,8,9,10};

Пример многомерного массива:

int[,] mas = new int[3,3]{{11,12,13},{21,22,23},{31,32,33}};

**Ступенчатый** **массив** – это **массив** **массивов**, в котором длина каждого **массива** может быть разной.

Пример: int[][] myArr = new int[4][];

myArr[0] = new int[4];

myArr[1] = new int[6];

myArr[2] = new int[3];

myArr[3] = new int[4];

// Инициализируем ступенчатый массив

for (; i < 4; i++)

{

myArr[0][i] = i;

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

}

Console.WriteLine();

for (i = 0; i < 6; i++)

{

myArr[1][i] = i;

Console.Write("{0}\t", myArr[1][i]);

}

Console.WriteLine();

for (i = 0; i < 3; i++)

{

myArr[2][i] = i;

Console.Write("{0}\t", myArr[2][i]);

}

Console.WriteLine();

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

{

myArr[3][i] = i;

Console.Write("{0}\t", myArr[3][i]);

}

Console.ReadLine();

}

Цикл foreach служит для циклического обращения к элементам коллекции, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла foreach:

foreach (тип имя\_переменной\_цикла in коллекция)

{оператор;}

Здесь тип имя\_переменной\_цикла обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла foreach. А коллекция обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, тип переменной цикла должен соответствовать типу элемента массива. Кроме того, тип может обозначаться ключевым словом var. В этом случае компилятор определяет тип переменной цикла, исходя из типа элемента массива. Это может оказаться полезным для работы с определенного рода запросами. Но, как правило, тип указывается явным образом.

Оператор цикла foreach действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными.

Существует два основных типа коллекций — универсальные и неуниверсальные коллекции. Универсальные коллекции являются строго типизированными во время компиляции. Таким образом, универсальные коллекции обычно обеспечивают более высокую производительность.

// 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

Кортежи предоставляют удобный способ для работы с набором значений.

Кортеж представляет набор значений, заключенных в круглые скобки:

var tuple = (5, 10);

Console.WriteLine(tuple.Item1); // 5

Console.WriteLine(tuple.Item2); // 10

tuple.Item1 += 26;

Console.WriteLine(tuple.Item1); // 31

**Свойство кортежа:**

►создается один раз и остается неименным (все свойства доступны только для чтения)

►позволяют использовать методы CompareTo, Equals, GetHashCode и ToString, свойство Size

►реализуют интерфейсы IStructuralEquatable, IStructuralComparable и IComparable (можно сравнивать)

**Локальные функции** — это новая возможность в C# 7, которая позволяет определять функцию внутри другой функции и вызывается только внутри этой функции.