КПИП. Урок 1

Введение

Цели и задачи учебной дисциплины. Связь с другими учебными дисциплинами. Перспективы развития технологий. Классификационная характеристика нынешних языков программирования.

Целью изучения дисциплины является формирование компетентности будущего специалиста в области разработки программных средств, использование языка программирования при реализации конкретных задач. Материал дисциплины Кпияп тесно связан с такими дисциплинами как: Оапи и Сиаод. В результате изучения на уровне мы должны знать перспективы развития технологий и создания программных средств, современные среды разработки программных средств для различных платформ. Знать на уровне понимания: методику создания программ, современную интегрированную среду разработки ПО.

Принципы ООП. ООП и процесс разработки программного обеспечения.

Разрабатывать классы и приложения с использованием ООП, создавать WIN-приложения, используя современную интегрированную среду разработки, организовывать доступ к БД из приложения, работать м XML-данными, создавать инсталяторы приложения, оформлять разработанные программные средства для дальнейшего внедрения и сопровождения.

Парадигмы программирования.

Парадигма программирования – это совокупность идей и понятий, определяющих стиль написания компьютерных программ. Виды парадигмы: Императивная или процедурная ПП (развилась на базе низкоуровневых языков, основанных на архитектуре Фон Неймана). Императивная программа состоит из последовательно выполняемых команд и вызова процедур, которые обрабатывают данные и изменяют значение переменных в программе, переменные при этом рассматриваются как некий контейнер для данных. Функциональная (состоит из набора взаимосвязанных, как правило рекурсивных функций. Каждая функция определяется выражением, которое задает правило вычисления ее значений в зависимости от значений ее аргумента. Выполнение функциональной программы заключается в последовательном вычислении значений функциональных вызовов). Логическая (программа рассматривается как множество логических формул, описывающих свойство некоторых объектов и теорем, которые необходимо доказать. Основные различия указанных парадигм касаются не только концепции программы, но и роли переменных. В отличии от императивных программ в функциональных и логических программах отсутствует явное присваивание значений переменных и как следствие побочные эффекты. Переменная в таких программах подобна переменным в математике. Особенность Ф и Л парадигмы — это использование рекурсии вместо циклов. Объектно-ориентированная парадигма. В ООП программа описывает структуру и поведение вычисляемых объектов и классов объектов. Объект обычно включает некоторые данные (состояние объекта) и операции с этими данными (методы) описывающие поведение объекта. Классы представляют множество объектов со схожей структурой и схожим поведением. Обычно описание классов имеет иерархическую структуру, включающую полиморфизм операций. Выполнение ООП представляет собой обмен сообщениями между объектами в результате которого они меняют свои состояния.

Свойства парадигм программирования

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Парадигма | Ключевой концепт | Программа | Выполнение программы | Результат |
| Императивная | Команда | Последовательность команд программы | Исполнение команд | Итоговое состояние памяти |
| Функциональная | Функция | Набор функций | Вычисление функции | Вычисление главной функции |
| Логическая | Предикат | Логические формулы | Логическое доказательство | Результат доказательства |
| О-О | Объект | Набор классов объекта | Обмен сообщения между объектами | Результирующее состояние объекта |

Поскольку парадигмы существуют и развиваются в рамках ЯП, которые призваны обеспечивать удобные механизмы решения различных прикладных задач неизбежна интеграция парадигм. Так уже в первых императивных алгоритмических языках, таких как Fortran, существовали элементы функционального стиля. Большинство современных языков аккумулируют в себе элементы и приемы нескольких стилей и тем не менее их можно классифицировать по основному им ядру: Императивная парадигма (паскаль), функциональная парадигма, логическая парадигма (пролог), О-О. Императивная и О-О: (делфи, ц++), функциональная и О-О, Логическая и О-О (обжект пролог). Сравнивая разные языки и парадигмы следует отметить существенное отличие следствия приемов традиционной императивной парадигмы программирования от средства приемов нетрадиц. Парадигмы. Языки программирования основанные на нетрадиционных парадигмах отличаются еще и методом реализации. Программы на таких языках обычно интерпретируются, а не компилируются как в Имп и ОИмп. Еще одно важное отличие они ориентированы на символьную обработку данных поскольку средства этих языков включают: древесные структуры данных такие как списки Лиспа, встроенный механизм сопоставления с образцом и бэктрекинга. Функционалы и функции высшего порядка и механизм частичных вычислений.

История C#

C# живет по принципу всякая сущность есть объект «Язык основан на строгой компонентной архитектуре и реализует передовые механизмы обеспечения безопасности кода. Сторонники С# называют его самым универсальным продвинутым и ахуенным ЯП так как основано на платформе микрософт дот нет. Название С# несет сакральный смысл. Знак решетка повышение. Автором этого языка программирования стали Скотт Валтамут и Андрес Хейльсберг, они также создатели турбо паскаля и дельфи. По одной из версий наши авторы вынашивали замысел нового языка и даже новой платформы которая сейчас носит название дот нет. С# поддерживает все 3 «столпа» ООП: инкапсуляция, наследование и полиморфизм. Кроме того в нем была реализована автоматическая «сборка мусора», обработка исключений и динамическое связывание.

Принципы ООП. ЯП C#. Платформа .net

ООП – это методология программирования, основанная на представлении программного продукта в виде совокупности объектов, каждый их которых является экземпляром конкретного класса. ООП используют в качестве базовых элементов взаимодействия объектов.

Объект – это именованная модель реальной сущности, обладающая конкретными значениями свойств и проявляющая свое поведение.

Объект – это обладающий именем набор данных (полей и свойств объекта) физически находящихся в памяти ПК и методах, имеющих доступ к ним. Имя используется для работы с полями и методами объекта. Любой объект относится к определенному классу. Объект – это конкретный экземпляр класса.

|  |
| --- |
| Hоuse- имя класса |
| -id:mt  -masonry:string набор полей с именами типами свойствами класса |
| +repair(): void список методов, их параметры и возвращающее значение |

ООП основано на принципах: инкапсуляции, наследования и полиморфизма, в частности позднего связывания.

Инкапсуляция – это принцип, объединяющий данные и код, манипулирующий этими данными, а также защищающий данные от прямого внешнего доступа и неправильного использования. Другими словами, доступ к данным класса возможен только посредством методов этого же класса.

Наследование – это принцип, посредством которого один класс может наследовать свойства другого класса и добавлять к ним свойства и методы характерные только для него. Наследование бывает двух видов:

1. Одиночное наследование – это подкласс, имеет один и только один суперкласс. Класс может иметь любое количество предков.

Полиморфизм – это механизм, использующий одно и тоже имя метода для решения похожих, но несколько отличающихся задач в различных областях, объектах при наследовании из одного суперкласса. Целью полиморфизма является использование одного имени при выполнении общих для суперкласса и подкласса действий.

Механизм «позднего связывания» в процессе выполнения программы определяет принадлежность объекта к конкретному классу и производит вызов методов, относящихся к классу, объект которого был использован. Механизм «позднего связывания» позволяет определять версию виртуального метода во время выполнения программы.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Объект подкласса может использоваться всюду где используется объект суперкласса. То есть при добавлении к иерархии классов нового подкласса существующий код с экземпляром нового подкласса будет работать точно также, как и со всеми другими экземплярами классов иерархии.

Платформа.нет фреймворк – это интегрированный компонент виндовс, который поддерживает создание и выполнение нового поколения приложений и веб-служб xml. Двумя основными компонентами платформы .net является общеязыковая среда выполнения CLR и библиотека классов .net framework.

Основные преимущества платформы .net

.net framework – это программная платформа для построения приложений на базе семейства ОС виндовс, а также на других ОС.

Ключевые свойства, поддерживаемые .net

1. Возможность взаимодействия с существующим кодом. Эта возможность позволяет комбинировать существенные двоичные компоненты технологии ком с более новыми программными компонентами .net и наоборот. С выходом .net 4.0 и последующих версий возможность взаимодействия дополнительно упростилась благодаря добавлению ключевого слова dynamie;
2. Поддержка многочисленных ЯП;
3. Общий исполняющий механизм, разделяемый всеми поддерживающими .net языками.
4. Языковая интеграция. В .net поддерживается языковое наследование, межязыковая обработка исключений и межязыковая отладка кода.
5. Обширная библиотека базовых классов. Данная библиотека позволяет избежать сложностей, связанных с выполнением низкоуровневых обращений API-интерфейса и предлагает согласованную объектную модель, используемую всеми поддерживающими .net языками.
6. Упрощенная модель развертывания. В отличии от com библиотеки .net не регистрируются в системном реестре. Более того платформа .net позволяет сосуществовать на одном и том же компутере нескольким версиям одной и той же сборке (сборка \*.dll).

Блоки платформы .net

Уровень использующей среды называется общеязыковой исполнительной средой (CLR). Главная задача CLR является автоматическое обнаружение, загрузка и управление объектами .net.

Общая система типов (CTS). В спецификации CTS полностью описаны все возможные типы данных и все программные конструкции, поддерживаемые исполнительной средой. Важно понимать, что отдельно взятый язык, совместимы с .net может не поддерживать все функциональные средства определенной спецификации CTS.

Поэтому существует общеязыковая спецификация (CLS). В данной спецификации описано подмножество общих типов и программных конструкций, которые должны поддерживать все ЯП для .net.

Роль библиотек базовых классов.

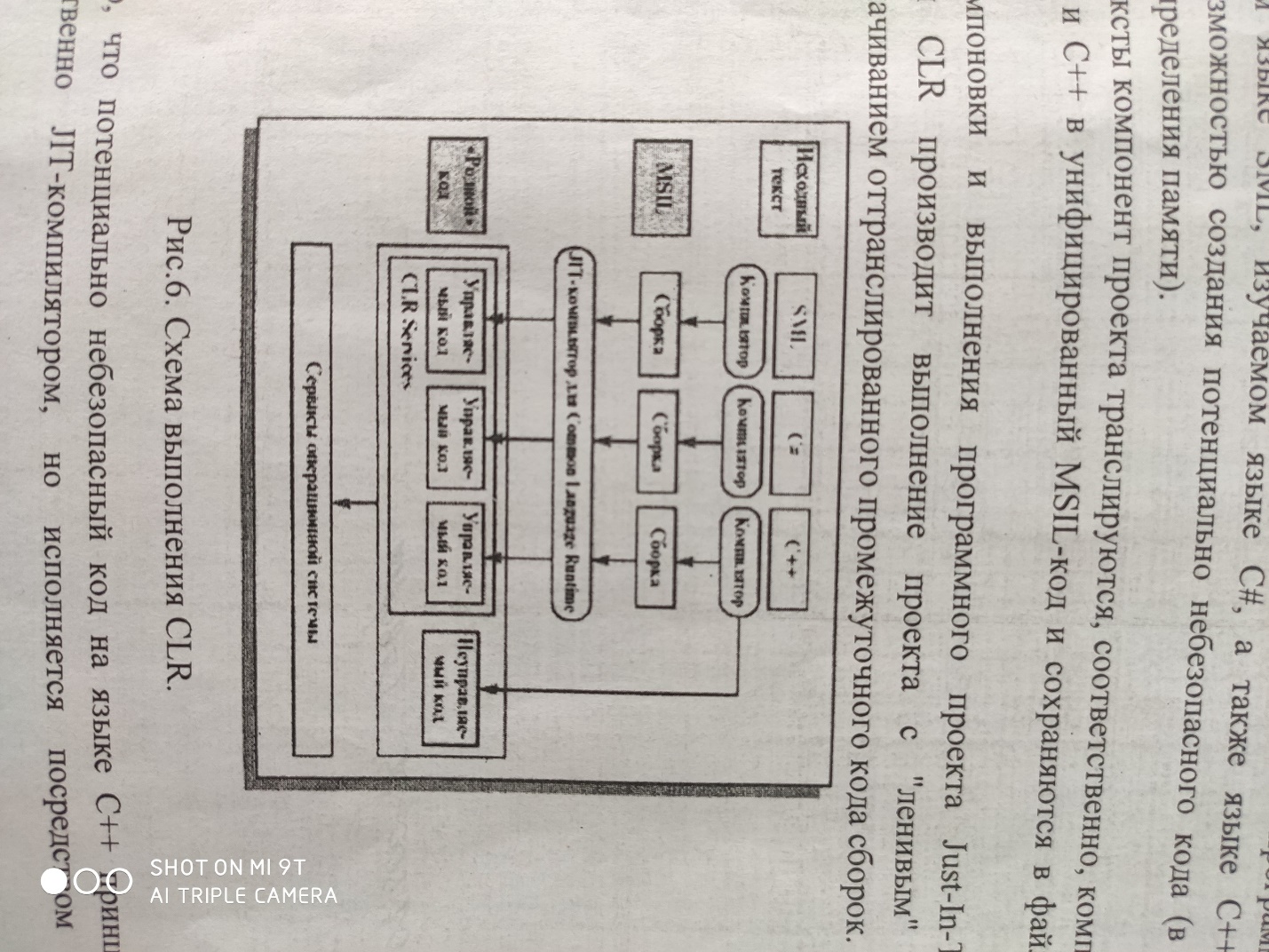
В дополнение к среде CLR и спецификациям CTS/CLS платформа .net предоставляет библиотеку базовых классов, которая доступна всем ЯП .net. Эта библиотека не только инкапсулирует разработанные примитивы такие как потоки, файловый ввод/вывод, системы визуализации графики и механизмы взаимодействия с различными внешними устройствами, но также обеспечивает поддержку для многочисленных служб требуемых большинством приложений. Библиотеки базовых классов определяют типы которые можно использовать для построения программных выражений любого вида. Кроме того, библиотеки базовых классов предоставляют типы для взаимодействия с XML-документами, локальным каталогом и файловой системой текущего компутера для коммуникации с революционными БД. На высоком уровне отношения между CLR, CTS, CLS и библиотеками базовых классов выглядит следующим образом.

1. Доступ к базам данных
2. Многопоточная обработка
3. API-интерфейсы для построения графических пользовательских интерфейсов настольных приложений
4. Файловые ввод/вывод
5. Безопасность
6. API-интерфейсы для web-приложений
7. API-интерфейсы для удаленной работы
8. Другие возможности

В .net существуют два типа библиотек: BSL и FSL.

BSL – это стандартная библиотека для всех языков платформы .net которая содержит набор пространств имен, внутри которых содержатся набор базовых типов, которые обеспечивают файловый и консольный ввод/вывод, безопасность, XML и сетевой доступ многопоточную обработку и другое. Главной сборкой данной библиотеки является файл mscorlib.dll. Вторым типо библиотек .net является библиотека классов платформ. Она является вторым основным компонентов платформы .net, которая упрощает работу программиста, предоставляя ему более расширенные библиотеки, такие как: ASP.NET, ADO.NET, WindowsForms и WPF.

.NETFramework=CLR+LiBraries(FCL,BCL).



Предположим, что компоненты проекта написаны на трех языках программирования. C++ характеризуется созданием потенциально опасного кода. Исходные тексты проекта транслируются собственными компиляторами с наших языков в универсальный MSIL код и сохраняется в файлах в виде сборок. В ходе компоновки и выполнения программного проекта G-компилятор среды CLL производит выполнение проекта с ленивым обозначиванием транслируемого промежуточного кодов. Существенно потенциально не безопасный код на языке C++ принципиально не выполним собственным же компилятором, но и используется и исполняется по средствам сервисов ОС. Существенным отличием микрософт дот нет от аналогов является универсальная система компиляции. В ходе программа на дот нет трансформируется в соответствии с заранее заданной обобщенной спецификацией языка (CTL). Система типов CTS полностью описывает все типы данных поддерживаемые средой выполнения, определяет их взаимосвязи и хранит их отображение в системе типов дот нет. Система типов микрософт дот нет образует иерархию с возрастанием снизу вверх, который явно выделяется две большие группы типов – это типы ссылки и типы значения. Различия между ними определяется особенностями вызова процедуры по имени или по значению и по ссылке. Система типизации микрософт дот нет позволяет пользователю создавать собственные типы как типы ссылки, так и типы значения на основе уже существующих.

Концепция интегрированной среды разработки приложений.

Интегрированная среда – это набор инструментов для разработки и отладки программ, имеющий общую интерактивную графическую оболочку, поддерживающую выполнение всех основных функций жизненного цикла разработки программы. Это набор и редактирование исходного текста, компиляция, исполнение, отладка, профилирование и тд. Одно из первых интегрированных средств стала среда турбопаскаль. Она была разработана в середине 80 годов.

История интегрируемых средств

Идея ИС достигла еще больше развития в середине 80 когда появились две группы популярных ИС: GNUEmacs. ИС для тех лет была среда Smalltalk. Турбо среды оказали огромное влияние на разработчиков программного обеспечения и создателей инструментов разработки ПО. Их характерной чертой стала поддержка непрерывного цикла разработки: набор и редактирование исходного текста, компиляция, анализ и исправление ошибок, завершение компиляции. Самым важным нововведением в среду турбопаскаль было расширение входного языка ОО концепциями (класс объекта) и конструкциями UNIT (модули), воплощая в себе идею независимой единицы компиляции.

Основные возможности современных ИС.

Характеризуются след компонентами: Единая интегрированная оболочка, обеспечивающая вызов всех других компонентов, не выходя из среды с широким использование функциональных клавиш; текстовый редактор для набора и редактирования исходных текстов программ; Система поддержки сборки то есть компиляция проектов из исходных кодов, включающая компилятор с исходного реализуемого языка и компоновщик объектных бинарных кодов в единый исполнительный код; отладчик – для отладки программ. В современных версиях появились еще несколько возможностей:

1. Профилировщик – это инструмент для накопления и анализа статических данных, полученных в результате исполнения программы под управление интегрируемой среды (число вызовов процедур, объем памяти используемых при выполнении программы).
2. Рефакторинг – это инструментальный систематических модификаций программ в среде без принципиальных изменений их функциональности с целью улучшения кода.
3. Генератор тестов – это инструмент для генерации типовых тестов для тестирования модулей, методов или процедур с различными возможными сочетаниями значения аргументов.
4. Система управления версиями исходных кодов или инструмент интеграции среды с одной из существующих версионных систем (CVS, RCS, Mercurial).
5. Инструменты поддержки командной разработки программы.
6. Инструменты анализа кода.
7. Инструменты визуализации генерируемого бинарного кода.
8. Инструменты «запутывания» кода, выполняющие именно с этой целью замену имен элементов кода (классов, методов, полей) на непонятные имена с целью затруднения изучения нашего кода.
9. Поддержка создания различных видов программных проектов и решения на основе типовых шаблонов кода.
10. Поддержка моделирования структуры программ на языке моделирования UML. Современные интегрируемые среды поддерживают языка UML: генерация модели и соответствующие диаграммы по исходному коду и наоборот генерация исходного кода по разработанной модели.

Моно языковые и много языковые интегрируемые среды.

Первоначально интегрируемые среды разрабатывают для программирования на каком либо одном исходном языке. Постепенно появлялась тенденция по превращению таких моно языковых интегрированных сред в много языковые. Поскольку для разработки проекта на различных языках используются исходные принципы и механизмы. Кроме того удобно использовать в большем проекте фрагменты программы написанной на разных языках. Среда визуал студио дот нет с самого начала создавалась как много языковая среда. Это принципиальная установка фирмы микрософт дать возможность разработчикам выбрать наиболее удобный язык или языки для соответствующих частей разработанного проекта, а затем собрать проект из бинарных компонентов, полученных путем компиляции с соответствующих языков единый бинарный промежуточных код CIL.

Типы данных.

Переменные типа значения содержат значения, а в переменных ссылочного тип хранятся ссылки на данные. Для ссылочного типа переменных допустимо, чтобы две переменные ссылались на один объект. Таким образом действие над одной переменной влияет на объект, на который ссылается другая переменная. В случае типа значения каждая переменная имеет собственную копию данных, и операция над одной переменной не может повлиять на другую.

Переменная – это сущность, значение которой может изменятся. В С# переменная – это область компьютерной памяти, которая идентифицируется и используется для хранения значений. Данные которые записаны в переменную называются значением этой переменной. В C# память под переменные выделяется в момент их создания.

Переменной в момент создания присваивается имя, которое уникально идентифицирует переменную в ее области видимости.

Переменную можно инициализировать после создания.

Когда вы обращаетесь с переменной, то мы обращаемся к значению, хранящемуся в этой переменной.

Тип имя переменной;

Переменная = значение;

Типы значения:

1. Хранят актуальное значение в стеке;
2. Значения могут быть любого встроенного или определенного пользователем типа данных.

Ссылочные типы:

1. Хранят адреса размещения в памяти других переменных;
2. Значения могут принадлежать любому встроенному или пользовательскому типу данных.

Ссылочные типы

Объект – это встроенные ссылочный тип данных. Базовый класс для все предопределенных и пользовательских типов данных.

Строка – это встроенный ссылочный тип данных, которые представляет строку символов Unicode. Также позволяет присваивать и изменять строковые значения. Однажды созданная строка не может быть модифицирована.

Класс – это определенная пользователем структура, содержащая переменные и методы.

Делегат – это определенный пользователем тип, содержащий ссылку на один или более методов.

Интерфейс – это тип определённого пользователем класса, которые используется для сложного наследования.

Массив – это определенная пользователем структура, содержащая значение одного типа данных.   
Структурные типы данных хранятся в стеке.

Ссылочные типы данных хранятся в куче.

Ключевое слово var

Которое позволяет создавать переменные без явного указания типа данных (в 3 версии). Тип данных такой переменной определяет компилятор по контексту инициализации. Var сохраняет принцип строгой типизации в C#.

Ключевое слово var имеет ограничение по его использованию:

1. Не может быть полем класса;
2. Не может быть аргументом функции;
3. Не может быть возвращаемым типом функции;
4. НЕ может быть переменной, которой присваивается null.

Преобразование встроенных типов данных.

Неявное (автоматическое).

Явно (ручное).

Ограничение для переменных с неявной типизацией:

* Нельзя сначала объявить неявно типизируемую переменную, а затем инициализировать.
* Нельзя указать в качестве значения неявно типизированной переменной null.

При объявлении переменных также нужно учитывать, что в одном контексте нельзя определить несколько переменных с одним и тем же именем.

Преобразование строки.

Метод parse () в качестве параметра принимает строку и возвращает объект текущего тип. Стоит отметить, что парсинг дробных чисел зависит от настроек текущей культуры. Чтобы не зависит от культурных различий можно установить четкий формат с помощью класса NumberFormatInfo и его свойства NumberDecimalSeparator. При использовании метода parse можно столкнуться с ошибками, например, при передаче алфавитных символов вместо числовых и в этом случае более удобным выбором будет применение метода TryParse. Данный пытается преобразовать строку к типу и если преобразование произошло успешно, то возвращает значение true иначе false. Также класс convert представляет еще один способ для преобразования значений. Для этого в нем определены следующие статические методы: Toboolean, ToChar. В качестве параметра в эти методы может передаваться значение различных примитивных типов, необязательно строки.

Структура программы. Переменные, операции и выражения.

Язык C# требует, чтобы вся логика программы содержалась внутри определения тип. В отличии от многих других языков в C# не допускается создание глобальных функций и глобальных элементов данных. Вместо этого все данные члены и все методы должны содержаться внутри определения тип. Весь код, который будет создан в рамках любого проекта будет написан внутри классов или структур – это и есть определение типа.

//подключаемые библиотеки

Using system;

//пространство имен программы

Namespace ConsoleApplication

{

//основной класс

Class Program

//точка входа

Static void Main(string[] args)

{

//инструкции

}

}

}

Подключение библиотеки – здесь указывается пространство имен из библиотек, которые используются в программе.

Пространство имен программы – используется для упорядочения классов п проекте, является не обязательным.

Основной класс – синтаксис языка C# требует создание хотя бы одного класса для корректной компиляции программы.

Точка входа – метод Main является точкой входа в программу, с него начинается выполнение программы написанных для платформы .net.

Инструкция – то что мы должны выполнить.

Приложение в C# должно содержать класс, определяющий метод Main. Формально класс, который определяет метод Main называется объектом приложения, хотя в одном исполняемом приложении разрешено иметь несколько объектов приложений. Программисту потребуется проинформировать компилятор о том какой из методов Main должен использоваться в качестве точки входа. Static это область действия статических членов Охватывает область класса, а не уровень объекта. Поэтому они могут вызываться без предварительного нового экземпляра класса.

Метод Main принимает один параметр, которые представляет собой массив строк. Хотя в текущий момент этот массив никак не обрабатывается. Данный параметр может содержать любое количество входных аргументов командной строки. Метод Main конфигурирован с возвращающим значением void, которое означает, что мы не определяем явно возвращаемое значение с помощью ключевого слова return перед выходом из области действия метода.

Директива Using.

Слово Using подключает необходимые библиотеки к проекту. Если вы решили использовать функции работы с файлами, то вам необходимо подключить System.IO. Среда визуал студио сама подскажет какого пространства имен не хватает для работы и предложить добавить его в код проекта.

Операция:

1. Унарная.
2. Бинарная.
3. Тернарная.

Бинарные арифметические операции.

Унарные операции

++ операция инкремента. (декремент также только с минусом).

Инкремент бывает префиксным – сначала значение переменной x увеличивается на единицу, а потом ее значение возвращается в качестве ее результата. (++x).

Инкремент бывает постфиксным - сначала значение возвращается в качестве ее результата, а потом значение переменной x увеличивается на единицу. (x++).

При выполнении сразу арифметических операций следует учитывать порядок их выполнения. Приоритет операции от наивысшего к низшему. (от инкремента до сложения).

Тернарная операция

Ассоциативность операторов

Типы операторов:

Левоассоциативные операторы, которые выполняются слева направо

Правоассоциативные операторы, которые выполняются справа налево

Все арифметические операторы, кроме префикса являются левоассициативными.

Операторы Отношения.

Логические отношения

Логическое умножение &

Умножение производится по разрядно. И если у обоих операндов значение разряда равно 1, то операция возвращает единицу, иначе число 0.

Логическая отрицание или инверсия (казюлька).

Логическое сложение |

Логическое или ^

Если у нас значение текущего разряда у обоих числе равны, то возвращается 1, иначе возвращается 0.

Операции сдвига

Применятся к целочисленным операндам.

Сдвигают двоичное представление первого операнда влево и вправо на количество двоичных разрядов, заданное вторым операндом.

При сдвиге вправо освободившееся разряды обнуляются.

Оператор присваивания

Сборки. Атрибуты. Директивы. Пространство имен.

В результате компиляции приложение визуал студио или в консоли, результатом этой работы является файл exe или dll, который называется сборкой приложения. Сборки кристаллизуют всю библиотеку классов .net. При написании кода и создании сборки приложения используются пространства имен, которые размещены в других сборках .net. Сборки имеют следующие составляющие:

* Манифест, который содержит метод данной сборки;
* Метаданные типов, используя эти метаданные сборка определяет местоположение типов в файле приложения, а также место размещения их в папке и памяти;
* Собственно, код приложения на языке MSIL;
* Ресурсы.

Все эти компоненты могут находиться в одном файле и тогда сборка представляет собой один единственный файл формата exe или dll. Также может быть, что эти компоненты хранятся в отдельных файлах.

Манифест сборки

Ключевым компонентом сборки является ее манифест. Если у сборки отсутствует манифест, то заключенный в ней код выполнятся не будет. Манифест может находится в одном файле с исполняемым кодом, а может размещаться и в отдельном файле. Манифест хранит следующие данные:

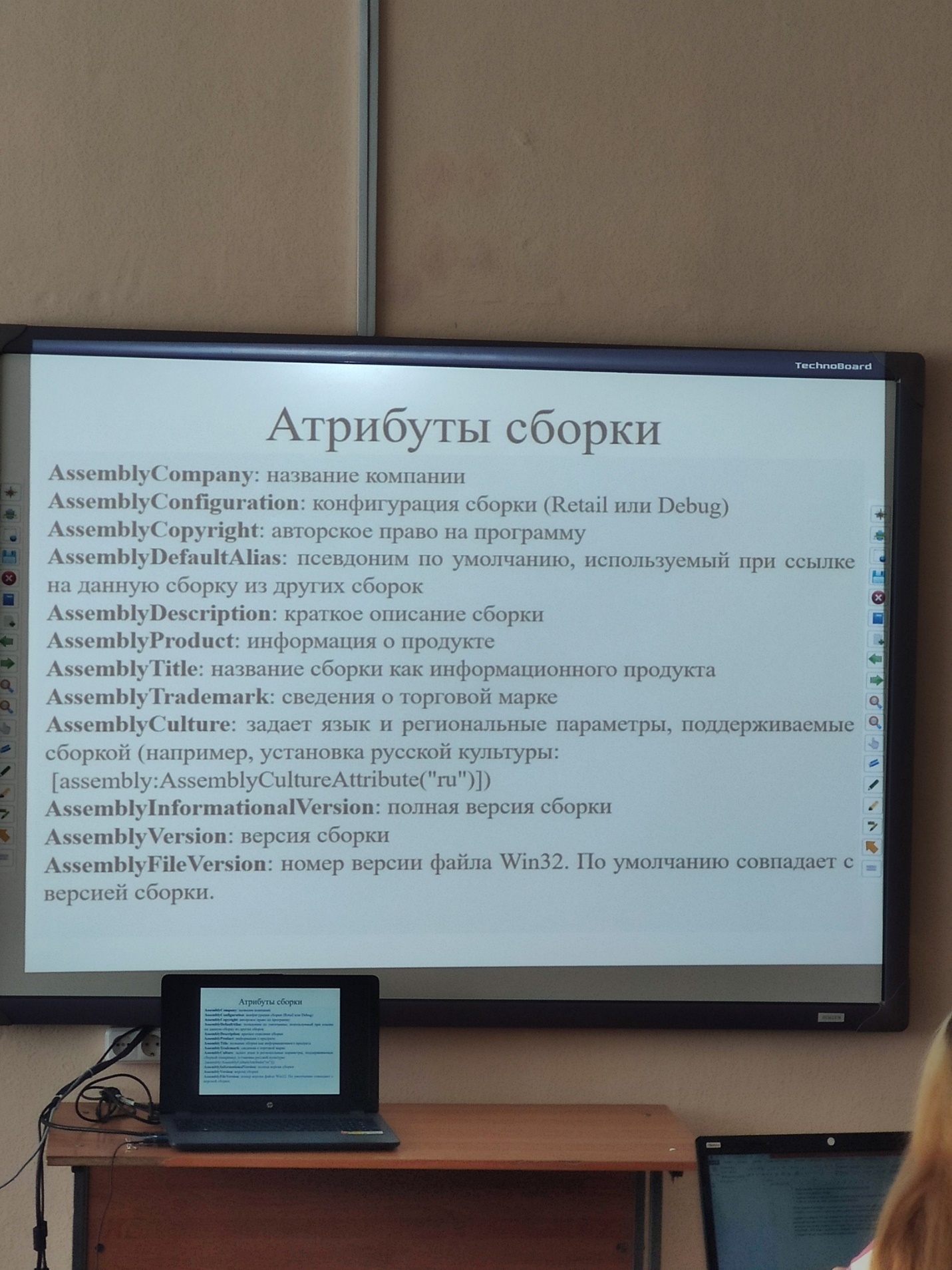
* Имя сборки;
* Номер версии;
* Язык и региональные параметры;
* Информация о строгом имени;
* Список всех файлов сборки;
* Список ссылок на другие сборки, которые используют текущая сборка;
* Список ссылок на типы, используемые сборкой.

Таким образом манифест позволяет системе определить все файлы, входящие в сборку, сопоставить ссылки на типы, ресурсы, управляет контролем версий.

Атрибуты сборки

По умолчанию визуал студио при создании проекта добавляет файл assemblyinfo.cs, который находится в properties.

Атрибуты сборки: смысл этого файла состоит в том, что он задает настройки манифесту сборки. Через атрибуты типа можно установить значение в манифесте. Префикс assembly перед атрибутом указывает на то, что это атрибут уровня сборки.



Также есть и графический способ определения информации о сборке.

Пространство имен

Каждая C# программа так или иначе использует некоторое пространство имен. C# автоматически предоставляет программе пространство имен по умолчанию. Но реальным программам придется создавать собственные или взаимодействовать с другими пространствами имен. Пространство имен определяет декоративную область, которая позволяет отдельно хранить множества имен. По существу имена, объявленные в одном пространстве имен не будут конфликтовать с такими же именами, объявленными в другом. Библиотека .net использует пространство имен System, поэтому в начале любой программы включена библиотека using.

Объявление пространства имен

Пространство имен с помощью ключевого слова namespace. Все что определено внутри пространства имен находится в пределах его области видимости. Внутри пространства имен можно объявлять классы, делегаты, структуры, интерфейсы, перечисления и другие пространства имен.

Пространство имен предотвращающие конфликтов по совпадению имен

Основное преимущество использования пространства имен состоит в том, что имена объявленные внутри одного из них не конфликтуют с такими же именами, объявленными вне его.

Существует две формы применения директивы using. В данной форме все члены определенные внутри заданного пространства имен становятся частью этого пространства , поэтому их можно использовать без дополнительного упоминания его имени. Во второй форме (using псевдоимя = имя) здесь элемент псевдоимя задает еще имя для класса или пространства имен, заданного элементом имя.

Директива препроцессора

В C# определен ряд директив предпроцессора, оказывающих влияние на интерпретацию исходного кода программы компилятором. Эти директивы определяют порядок е текста программы перед ее трансляцией в объектный код в том исходном файле где они появляются. Термин директива предпроцессора появилась в связи с тем, что подобные инструкции по традиции обрабатывались на отдельной стадии компиляции, называемой предпроцессором. Все директивы предпроцессора начинаются со знака #, кроме того каждая директива предпроцессора должна быть выделена в отдельную строчку кода.

Директива #define

Данная директива определяет последовательность символов, называемую идентификатором. Присутствие или отсутствие идентификатора может быть определено с помощью директивы #if и #endif и поэтому используется для управления процессором.

Общая форма директивы. #definе идентификатор

Директива #if и #endif, допускают условную компиляцию последовательности кода в зависимости от истинного результата вычисления выражения, включающего в себя один или несколько идентификаторов.

Идентификатор считается истинным если он определен, иначе ложь.

Общая форма #if идентификационное\_выражение

Последовательности операторов

#endif

Директивы #else и #elif

Действуют аналогично условному оператору в C#

Общая форма #elif

#if идентификационное выражение

Последовательность операторов

#elif идентификационное\_выражение

Последовательность операторов

#elif идентификационное выражение

Последовательность операторов

#endif

Директива #undef с помощью этой директивы удаляется определенный ранее идентификатор

Общая форма директивы #undef

#undef идентификатор

Директивы #error вынуждает компилятор прервать компиляцию. Она служит в основном для отладки.

Общая форма директивы #error

#error сообщение об ошибке

Когда в коде встречается директива выводится сообщение об ошибке

Директива #warning действует аналогично предыдущей, но выводит предупреждение, а не ошибку, а, следовательно, компиляция не прерывается

Общая форма директивы #warning

#warning предупреждающее сообщение

Директива #line задает номер строки и имя файла, содержащего эту директиву. Номер строки имя файла используется при выходе ошибок или предупреждений во время компиляции.

Общая форма директивы #line

#line номер “ имя файла “

#line default – данное ключевое слово обозначает возврат нумерации строк в исходное состояние.

#line hidden.

Директивы #region и #endregion c помощью данных директив определяется область, которая разворачивается или сворачивается при структурировании исходного кода в ИС разработки визуал студио

Общая форма

#region текст

Исполнительность кода

#endregion текст

Директива #pragma с помощью данной директивы инструкции задаются в виде опций.

Общая форма

#pragma опция

В C# предусмотрено две опции для данной директивы:

1. Warning – служит для разрешения или запрета отдельных нарушений со стороны компилятора и имеет две формы (disable и restore)

Для отмены используется 1 форма, а для разрешения 2.

1. Checksum - она служит для формирования контрольной суммы в проектах ASP.net

Общая форма #pragma checksum “имя файла” “{Cuid}” “контр”

Где имя файла обозначает имя файла, Cuid - Глобальный уникальный идентификатор, с которым связано имя файла, шестнадцатеричное число.

Управляющие операторы

Управляющие операторы языка высокого уровня:

* Следование
* Ветвление
* Цикл
* Передача управления

Блок – последовательность операторов, заключенная в операторные скобки

Блок воспринимается компилятором как один оператор и может использоваться всюду, где синтаксис требует одного оператора.

Пустой оператор; используется, когда по синтаксису оператор требуется, а по смыслу – нет.

Операторы ветвления

* Развилка (if)
* Переключатель (switch)

Условные оператор выполняет некоторый оператор если результаты вычисления выражения оказывается true.

If ( выражение ) оператор\_1;

[else оператор\_2;]

Изменение потока выполнения с помощью фигурных скобок.

Конструкция else всегда применяется к непосредственно предшествующему оператору if в блоке операторов.

В языке C# отсутствует аналог ключевого слово elseif.

Оператор switch

Операторы switch позволяют организовать ветвление пока выполнение программы на основе выбора из возможных значений, переменная может принимать. Операторы switch в результате могут дать более ясный ход чем множество операторов if потому что они требуют только однократной оценки выражения. Значения в каждом выражение касе должны быть константными, что ограничивает разрешенные типы встроенными целочисленными типами (bool, char, string). В конце каждой конструкции касе необходимо явно указывать куда выполнение должно передаваться дальше с помощью одного из операторов перехода (break, goto case x (переход на другую конструкцию else), goto default (переход на конструкцию default), return и тд).

Операторы цикла

Язык си шарп позволяет выполнять последовательность операторов, повторяющимся образом с помощью операторов while, do, for, foreach.

Цикл вайл многократно выполняет вход в своем теле до тех пор, пока результат вычисления выражения является труе. Выражение выполняется перед выполнением тела цикла.

While (выражение) оператор

Цикл с постусловием (do while)

Цикл ду вайл отличается по функциональности от цикла вайл только тем, что выражение в нем проверятся после выполнения блоков операторов (гарантируя, что блок оператора выполнится хотя бы один раз).

Цикл с параметром (for)

Цикл for похож на цикл вайл но имеет специальную конструкцию для инициализации и итерации переменных.

For (инициализация; условие; итерация)

Любая из трех частей оператора for может быть опущена.

Цикл foreach

Оператор foreach обеспечивает проход по всем элементам перечисляемого объекта.

Foreach (тип имя\_переменной\_цикла in коллекция) оператор;

Тип переменной цикла должен соответствовать типу элемента массива.

Передача управления или операторы перехода

* Break
* Continue
* Return
* Goto
* Throw

Оператор бреак – завершает выполнение цикла, внутри которого он записан

Оператор континуе – выполняет переход к следующей итерации цикла

Оператор ретурн – выполняет выход из функции, внутри которой он записан

Оператор гото – выполняет безусловную передачу управления

Оператор троу – генерирует исключительную ситуацию

Оператор бреак

Оператор бреак завершает выполнение тела итерации или оператора switch:

Int x = 0 while (true)

{

If (x++ > 5) break; // прекратить цикл

}

// после бреак выполнение продолжится здесь

Оператор континуе

Оператор континуе пропускает оставшиеся в цикле и начинает следующую итерацию

For (int i=0; i<10; i++)

{

If ( (1%2) == 0) continue;

Console.write (i+””); //13579

Оператор goto

1. Goto метка;

В теле той же функции должна присутствовать ровно одна конструкция виде: метка: оператор;

1. Goto case константое выражение;
2. Goto default;

StartLoop;

If ( I<=5)

{

Console.write (I + “ “); // 1 2 3 4 5

I + +;

Goto startLoop;

}

Оператор return

Оператор return завершает метод и должен возвращать выражение с возвращаемым типом метода, если метод не является воед;

Return [выражение];

Static decimal AsPercentage (decimal d)

{

Decimsl p = d \* 100;

Return p; // возвратиться в вызывающий метод со значение

Оператор ретурн может находиться в любои месте метода (кроме блоеп finally)

Процедуры и функции. Методы класса.

Процедуры и функции позволяют задавать определенную функциональность и многократно выполнять один и тот же программный код при различных значениях параметров. Долгое время процедуры и функции играли не только функциональную, но и архитектурную роль, но с появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ООП-языков в роле архитектурного модуля выступает класс. Процедуры и функции связываются с классом, и они обеспечивают функциональность данных классов и называются методами классов. Главную роль в программной системе играют данные, функции лишь служат там. В C# процедуры и функции существуют только как методы некоторого класса, и они не существуют вне класса.

Процедуры и функции отличны

Функция отличается от процедуры двумя особенностями: всегда вычисляет некоторое значение, возвращаемое в качестве результата функции, вызывается в выражениях. Процедуры в C# имеют свои особенности: возвращают формальный результат void, указывающий на отсутствие результата, вызов процедуры является оператором языка, имеет входные и выходные аргументы, причем выходных аргументов может быть достаточно много.

Определение методов

Если переменные хранят некоторое значение, то методы содержат собой набор операторов, которые выполняют определенные действия.

[модификаторы] тип возвращаемого значения название метода

{[параметры]}

{

// тело метода

}

Вызов методов. Для вызова метода указывается его имя, после которого в скобках идут его значения его параметров.

Название метода (значение для параметров метода);

Преимущество методов является то, что их можно повторно и многократно вызывать в различных частях программы.

Возвращение значение

Метод может возвращать значение какого-либо результата. Если метод имеет любой другой тип, отличный от воед, то такой метод обязан вернуть значение этого типа. Для этого применяется оператор return после которого идет возвращаемое значение.

Return возвращаемое значение;

Выход из метода

Оператор return не только возвращает значение, но и производит выход из метода, поэтому он должен определятся после всех операторов. Можно использовать r операторы. В этом случае после оператора return не ставится никакого возвращаемого значения.

Сокращенная запись методов.

Если метод в качестве тела определяет один оператор, то можно сократить определение метода.

Static void fadfas() => console.writeline(“Hello”)

Static void fadfas() => “hello”;

Если метод возвращает значение, то для того чтобы сократить запись метода используется => и возвращаемое значение оборачивается в круглые скобки.

Параметры методов

Параметры позволяют передавать в метод некоторые входные данные.

Значения, которые передаются параметрам называются аргументами. Формальные параметры – это собственно параметры метода. А фактические параметры — это значения, которые передаются формальным переменным. Если параметрами метода передаются значения переменных, которые представляют базовые примитивные типы, то таким переменным должно быть присвоено значение. При передаче значений параметрам важно учитывать тип параметров. Между аргументами и параметрами должно быть соответствие по типу.

Необязательные параметры

По умолчанию при вызове метода необходимо предоставить значения для всех его параметров, но C# позволяет использовать необязательные параметры. Для таких параметров необходимо объявить значения по умолчанию. Следует учитывать после необязательных параметров все последующие параметры также должны быть необязательными. Так как последние два параметра необязательные при вызове метода мы можем или объявить один, или опустить два.

Именованные параметры

При вызове методов значение для параметров передается в порядке объявления этих параметров методом, но можно и нарушить порядок использования используя именованные параметры.

Передача параметров в метод по ссылке

В C# значения переменных по умолчанию передаются по значению (в метод передается локальная копия параметра, который используется при вызове). Это означает что мы не можем внутри метода изменить параметр из вне. Чтобы передавать параметр по ссылке и иметь возможность влиять на внешнюю переменную мы используем два ключевые слова ref и out. В результате все изменения параметра в теле метода будут сохранены. Для работы с параметром ref определение метода и вызывающий метод должны явно использовать ключевое слово ref.

Особенности использования ref

* Перед передачей в метод переменную обязательно нужно инициализировать;
* В методе параметру можно не присваивать новое значение.

Ключевое слово out.

Тоже самое что и ref только перед передачей в метод переменную можно не инициализировать. В методе параметру обязательно присвоить новое значение.

Входные параметры. Модификатор in

Модификатор in указывает, что через данный параметр будет передаваться по ссылке, однако внутри параметра его значение параметра нельзя будет изменить.

Массив параметров в ключевое слово params

В предыдущих ситуациях мы использовали постоянное число параметров, но используя ключевое слово парамс мы может передавать неопределенное количество параметров. Сам параметр с ключевым словом парамс при определении метода должен представлять одномерный массив того типа данных, который мы собираемся использовать.

Если же нам надо передать какие-то другие параметры, то они должны быть указаны перед параметром с ключевым словом params.

Класс.

Класс определяется с помощью ключевого слава class.

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

Объявление экземпляра класса

Demo a = new demo ().

Данные: поля и константы

Объявление поля.

[атрибут] [спецификатор] [const] тип имя

[= начальное значение]

Обращение к статическому полю и обращение к константам осуществляется через имя класса.

Ключевое слово this

Каждый объект содержит свой экземпляр полей класса

Параметр this хранит ссылку на вызывающую функцию экземпляра. В явном виде параметр this применяется для того, чтобы возвратить из метода ссылку на вызывающий объект, а также для идентификации поля в случае, если его имя совпадает с именем параметра метода.

Свойства класса – это член класса, который осуществляет доступ к полю класса (чтение поля или запись).

Метод доступа get должен заканчиваться на операторе return.

Метод доступа set возвращает значение void. В нем используется неявный параметр value.

Массивы в C#

Массивом называют упорядоченную последовательность элементов одного типа. Каждый элемент массива имеет индексы, определяющие порядок элементов.

Массивы бывают статические и динамические.

Ко всем массивам относятся следующие замечания: каждый элемент массива содержит некоторое значение; элементы доступны по индексам, которые начинаются с 0; длинной массива называется общее кол-во элементов, которое он может содержать; нижняя граница массива — это индекс его 1 элемента; словом ранг принято называть размерность массива; Массивы бывают одномерные, многомерные и ступенчатые.

Объявлением массива указывает тип данных, который он будет хранить и идентификатор.

Объявление массива не выделяет для него памяти.

Объявление массива

Type[} arrayname;

Инициализация массивов

Массив может быть создан с помощью ключевого слова new. А затем инициализирован.

Массив может быть инициализирован во время объявления, в этом случае ключевое слово new не используется.

Создание и инициализация массива ключевым словом new включает создание размерности массива.

Создание массива

1 способ

Arrayname = new type [size-value];

2 способ

Type[] arrayname = new type [size-value];

Объявление и инициализация вообще могут быть размещены в разных операторах. Но в этом случает без операции new нельзя.

Int[] myarray;

Myarray = new int[] {1,3,5,7,9}; // так можно

Myarray = {1,3,5,7,9}; // так нельзя

Фактически имя массива является ссылкой на область динамической памяти, в которой последовательно размещается набор элементов определенного типа.

Для генерации элементов массива в C# используется класс Random. Чтобы им воспользоваться необходимо создать новый объект класса.

Random <имя объекта> = new Random();

Символы и строки постоянной длины

В C# есть символьный класс char. Он использует двух байтовую кодировку Unicode. Для этого типа в языке определены символьные константы – символьные литералы. Константу можно задать: символом, заключенным в одинарные кавычки, escape-последовательностью, Unicode-последовательностью.

Char ch=new Char();

переменная ch объявляется в объектном стиле, используя ключевое слово new и вызов конструктора классов. Явные или неявные преобразования между классами char и string отсутствуют. Но существует метод tosntring который позволяет переменные типа char преобразовать стандартным образом в тип string. Также существуют неявные преобразования типа char в целочисленные типы. Обратное преобразование целочисленных типов в тип char также существуют, но они являются явными.

Класс char[] – массив символов

На языке C# определен класс char[], и его можно использовать для представления строк постоянной длины. Его нельзя инициализировать строкой символов. Константа, задающая строку символов, принадлежит классу string а в C# не определены взаимные преобразования между классами string и char в виде массива, даже явное. Однако у класса string есть динамический метод ToCharArray, задающий подобное преобразование. Также возможно по символьно передать содержимое переменной string в массив символов. Класс Char[], как и всякий класс массив в C# является наследником нее только класса object, но и класса array и следовательно обладает всеми методами родительского класса.

Char[] stM1=”Здравствуйте”; неправильно

String hello= “ здраdствуйте”

Char[] stM1 = hello.ToCharArray(); правильно

Класс System.string

Char[] a={‘a’, ‘b’, ‘c’, ‘d’, ’c’,};

String v =new string (a,0,2)

В C# есть два вида строковых констант: обычные и собачковые (@) константы.

Неизменяемость объекта класса string.

String является не изменяемым типом данных. Это означает что однажды инициализированный строковый объект уже не может быть изменен. Методы и операции, модифицирующие содержимое строк создают новые строки.

Изменяемые строки

Чтобы создать строку, которую можно изменять в C# предусмотрен такой класс, как stringbuilder. Объекты этого класса всегда объявляются с явным вызовом конструктора класса (new).

StringBuilder a =new

StringBuilder();

По умолчанию если мы создаем пустую строку, то ее размерность 16 символов.

Для строк класса stringbuider вводится понятие емкости – выделенного объема памяти.

Регулярные выражения

Регулярным выражением называется правило обработки строк, представленное в виде так называемого шаблона (pottetn)

Для использования регулярных выражений в библиотеке .NET есть классы, которые объединены в пространство имен System.text.regular.Expressions.

Это пространство имен не добавляется автоматически.

Действия над строками

С помощью регулярных выражений можно:

* Поиск соответствий строке шаблона;
* Извлекать данные, содержащиеся в строках;
* Преобразовать формат строк;
* Кодировать и декодировать строки.

Шаблон регулярного выражения

* Шаблоном регулярного выражения называется специальная строка символов, задающая правило обработки обычных строк.

Регулярное выражение состоит из символов двух видов: обычных символов, представляющих в выражениях самих себя, метасимволов.

Метасимвол – это специальный символ, который служит обозначение класса символов (например, цифры или буквы), уточняет позицию поиска или задает количество повторений символов в выражении.

Регулярное выражение записывается в виде строкового литерала, перед строкой необходимо ставить символ @.

Символ @ можно не ставить, если в качестве шаблона используется шаблон без метасимволов.

Если нужно найти какой-то символ, который является метасимволов, можно это сделать, защитив ее обратным слешем. Т.е. просто означает любой одиночный символ, а \. Означает просто точку.

Классы символов.

Различат следующие классы символов:

* Положительные группы символов;
* Отрицательные группы символов;
* Любой символ;
* Символ пробела;
* Символ, не являющийся пробелом.

Точка – любой символ, кроме \n.

Последовательность символов или диапазон символов – любой одиночный символ из последовательности внутри скобок.

Например, выражение «К[иоэ-я]т» - соответствует фрагментами строк Кот, кит, кэт, кят или кют.

^диапазон символов – любой одиночный символ, не входящий в последовательность внутри скобок.

Например, выражение «К[иоэ-я]т» соответствует фрагментам строк Кfт, К&т.

\w – любая буква или цифра

Например, выражение «К\\wt» или @ «К\wt»

Соответствует фрагментами строк Кfт, К7г, Кот, К\_т

\s – любой пробельный символ (пробел, \n, \t, \w, \f, \r.

\S – любой не пробельный символ.

\d – любая десятичная цифра.

Например, выражение «К\\dt» или @ «К\dт» соответствует фрагментам строк К0т, К3т.

\D – любой символ, не являющийся цифрой.

^ - должен располагаться в начале строки.

$ - должен располагаться в конце строки.

\b – фрагмент, соответствующий регулярному выражению, должен располагаться на границе слова.

\*-ноль или больше повторений предыдущего элемента.

Классы пространства RegularExpressions

Ограничим наше рассмотрение следующими наиболее важными классами пространства имен RegularExpressions:

* Regex
* Match и MatchCollection

Класс Regex непосредственно реализует механизм регулярных выражений

Класс Regex

* Для описания регулярного выражения в классе определенно несколько перегруженных конструкторов

Regex() – создает пустое выражение;

Regex(RegexOptions) – создает заданное выражение и задает параметры для его обработки с помощью элементов перечисления RegexOptions (например, различать или нет прописные и строчные буквы).

Объекты класса RegEx

* Обрабатываемые строки, которые называют входными, передаются методам класса Regex в качестве параметров, а результатами обработки совпадения (match) – подстроки или коллекции подстрок исходной строки, удовлетворяющей шаблону.

Regex r = new Regex (@”\d{2,3} (-\d\d){2}”);

Структуры и перечисления

Структура – это значимый тип данных, экземпляр структуры размещается в стеке, а не в динамической памяти.

Синтаксис структуры:

[спецификаторы] struct <имя> [:интерфейсы]

{тело структуры}

Спецификаторы – public, internal, для вложенных структур можно private.

Правила описания структуры:

Структуры не могут участвовать в иерархиях наследования, но может реализовывать интерфейсы

В Структуре нельзя определить конструктор без параметров.

Конструктор по умолчанию автоматически определяется для всех структур, его нельзя изменить.

В структуре нельзя определить деструкторы.

Структура не может быть абстрактной и не может содержать абстрактные или виртуальные методы.

Объект структуры можно создать с помощью операции new, подобно любому объекту класса.

<имя структуры> <имя экземпляра>= new <имя структуры,>()

Можно объявить структуру не используя new.

Например

Student St3; St3.fam = “Петров”;

При присваивании одной структуры другой создается копия этого объекта.

Преимущества использования структур: структуры обрабатываются напрямую, а не через ссылки. Таким образом, структура не требует отдельной ссылочной переменной. Т.е. при использовании структур расходуется меньший объем памяти.

Перечисления

Перечисление – это множество именованных целочисленных констант.

Синтаксис

[спецификаторы] enum <имя> [:базовый тип]

{тело перечисления}

Допускаются спецификаторы new, public, protected, internal, private.

Базовый тип – это тип элементов, из которых построено перечисление. По умолчанию int.

Тело перечисления состоит из имен констант, который может быть присвоено значение, разделенных запятыми.

Описывать перечисление можно как в пространстве имен, так и внутри класса или структуры.

При использовании переменных перечисляемого типа в целочисленных операциях и выражениях требуется явное преобразование типа.

С переменными перечисляемого типа можно выполнять арифметические действия, логические поразрядные операции, сравнивать их с помощью операций отношения.

Все перечисления являются потомками базового класса System.Enum.

Приведем описание некоторых методов этого класса.

Статический метод

GetNsme (Type v, object v)

Возвращает строку- имя константы по ее значению.

GenNames (type t)

Возвращает строкой массив из имен констант, составляющих перечисление.

getValues (type t)

возвращает массив значений констант, составляющих перечисление.

Isdefined (Type t, object v)

Возвращает значение true, если параметр v содержит:

* Значение константы, входящей в перечисление.

Enum.Parse (type t, string s)

В пространстве имен System определены стандартные перечисления ConsoleColor и ConsoleKey.

Наследование

Класс может быть унаследован от другого класса с целью расширения или настройки исходного класса. Наследование от класса позволяет повторно использовать функциональность данного класса вместо его построения с нуля. Класс может наследоваться только от одного класса, но сам может быть наследован множеством классов, формируя иерархию классов. Класс от которого произошло наследование, называется базовым или родительским. Классы, которые произошли от базового, называются потомками или наследниками, или производными классами.

Виды наследования:

* От простого класса;
* От абстрактного класса;
* От интерфейса.

Производный класс наследует от базового класса все, что он имеет. Другое дело, что воспользоваться в производном классе можно не всем наследством.

Синтаксис

[атрибуты] [спецификаторы] class имя\_класса [:базовые способы]

{тело класса}

Конструкторы и наследование

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы (созданные программистом или системой).

Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса без параметров.

Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются, начиная верхнего уровня.

Интерфейсы

Интерфейс – это еще один инструмент для реализации полиморфизма в C#. Интерфейс представляет собой набор методов (свойств, событий), реализацию которых должен обеспечить класс, который реализует интерфейс. Основная идея использования интерфейса состоит в том, чтобы к объекту таких классов можно было обращаться одинаковым образом. Каждый класс определяет элементы интерфейса по-своему.

Синтаксис

[атрибуты] [спецификаторы] interface имя [: предки]

Тело интерфейса [:]

(имя начинается с i)

Интерфейс может наследовать нескольких интерфейсов, в этом случае предки перечисляются через запятую.

Тело интерфейса составляет абстрактные методы, шаблоны свойств и индексаторов, а также события.

Интерфейс не может содержать поля, операции, конструкторы, деструкторы, типы и любые статические элементы.

Отличия интерфейса от абстрактного класса

Элементы интерфейса по умолчанию имеют спецификатор доступа public и не могут иметь спецификаторов, заданных явным образом.

Интерфейс не может содержать полей и обычных методов – все элементы интерфейса должны быть абстрактными.

Класс, в списке предков которого задается интерфейс, должен определять все его элементы, в то время как потомок абстрактного класса может не переопределять часть абстрактных методов предка.

Класс мотет иметь в списке предков несколько интерфейсов, при этом он должен определять все их методы.

Реализация интерфейса

В С# поддерживается одиночное наследование для классов и множественное – для интерфейсов.

Сигнатуры методов в интерфейсе и реализации должны полностью совпадать.

Для реализуемых элементов интерфейса в классе следует указать спецификатор public.

К элементам можно обращаться как через объект класса, так и через объект типа соответствующего интерфейса.