**06.09.19**

**Программа** – это данные, предназначенные для управления конкретными компонентами системы, обработки информации в целях реализации определенного алгоритма.

**Программное обеспечение** – это совокупность программ, системы обработки информации и программных документов, необходимых для эксплуатации.

**Свойства программного обеспечения:**

1. Необходимость документирования.

2. Эффективность программного обеспечения

3. Надёжность.

4. Возможность сопровождения.

**Системная программа** – программа, предназначенная для поддержания работоспособности системы, обработки информации или повышения эффективности её использования.

**Прикладная программа** – это программа, предназначенная для решения задачи или класса задач в определенной области применения Системы Обработки Информации (СОИ).

**Промежуточное ПО** – это совокупность программ, осуществляющих управление вторичными ресурсами, ориентированными на решение определенного класса задач.

**Управляющая программа** – это системная программа, реализующая набор функций управления, которые включают в себя управление ресурсами и взаимодействие с внешней средой СОИ, востоковедение работы системы после проявления неисправности в технических средствах.

**Система программирования** – это система образуемая языком программирования, компилятором или интерпретатором программ, представленными на этом языке соответствующей документацией, а так же вспомогательными средствами для подготовки программ к форме, пригодной для выполнения.

**Программный модуль** – это программа или функционально завершенный фрагмент программы, предназначенный для хранения, трансляции, объединения с другими программными модулями и загрузки в ОП.

**Исходный модуль** – программный модуль на исходном языке, обрабатываемый транслятором и представляемый для него как целая, достаточная для проведения трансляция.

**Трансляция** – это преобразование программы, предоставленной на одном языке программирования в программу на другом языке программирования в определённом смысле, равносильную первой.

**Автокод** – это символьный язык программирования, по своей структуре предложение которого в основном подобны командам и обрабатываемым данным конкретного машинного языка.

**Объектный модуль** - программный модуль, получаемый в результате трансляции исходного модуля.

**Загрузочный модуль** – программный модуль, представленный в форме пригодной для загрузки в ОП для выполнения.

**Интерпретация** – реализация смысла в виде некоторого синтаксически законченного текста, представленная на каком-то языке.

**09.09.19 – 10.09.19**

**Концепция разработки программного модуля. Понятие жизненного цикла программы.**

Жизненный цикл программного обеспечения - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим жизненный цикл программного обеспечения, является международный стандарт ISO/IEC 12207.

Структура жизненного цикла базируется на трех группах процессов:

1. Основные процессы (приобретение, поставка, разработка, эксплуатация и сопровождение).

2. Вспомогательные процессы (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем).

3. Организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого жизненного цикла, обучение).

Разработка включает в себя: все работы по созданию программного обеспечения и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала. Разработка программного обеспечения включает в себя анализ, проектирование и реализацию-программирование.

Эксплуатация включает в себя: работы по внедрению компонентов программного обеспечения в эксплуатацию, конфигурирование базы данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение обучения персонала, непосредственно эксплуатацию в том числе локализацию проблем и устранение причин их возникновения, модификацию программного обеспечения в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает: выбор методов и инструментальных средств для реализации проекта, определение методов, описание промежуточных состояний разработки проекта, разработку методов и средств испытаний ПО, обучение персонала. Обеспечение качества проекта связано с проблемами проверки соответствию требованиям на данном этапе проектирования и тестирования ПО. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла, прежде всего процессы разработки и сопровождения ПО. При создании проектов сложных ИС, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций. Создание унифицированной структуры и обеспечение развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях жизненного цикла.

Общие принципы и рекомендации конфигурационного учета и планирования управления конфигурациями отражены в проекте стандарта ISO 1220 – 2.

Модели жизненного цикла:

1. Каскадная модель. Основной характеристической особенностью каскадной модели является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем. (схема: 1. анализ, 2. проектирование, 3. реализация, 4. внедрение, 5. сопровождение). Каждый этап завершается выпуском полного комплекта документации.

2. Спиральная модель. Делает упор на начальные этапы: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО. На нем уточняются цели и характеристики проекта, определяется его качество, и планируются работы следующего витка спирали. САНЯ. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная задача – можно быстро показать пользователям работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Основная проблема данного метода – не нарушать временные ограничения на каждый из этапов жизненного цикла.

**12.09.19**

**Современные методы разработки ПО.**

Метод нисходящего проектирования (метод пошаговой детализации, метод иерархического проектирования, top-down подход). Суть метода заключается в определении спецификаций компонентов системы путем последовательного выделения в ее составе отдельных составляющих и их постепенной детализации до уровня, обеспечивающего однозначное понимание того, что и как необходимо разрабатывать и реализовывать. Реализация метода нисходящего проектирования тесно связана с другим понятием программирования – модульное проектирование. Модуль – последовательность логически взаимосвязанных фрагментов задачи, оформленных как отдельная часть программы.

Свойства модулей:

1. На модуль можно ссылаться.

2. По завершении работы модуль должен возвращать управление тому модулю, который его вызвал.

3. Модуль должен иметь один вход и один выход.

4. Модуль должен иметь небольшой размер.

Преимущества модульного принципа:

1. Большую программу могут разрабатывать одновременно несколько исполнителей, что позволяет сократить сроки разработки.

2. Появляется возможность создавать и многократно использовать в дальнейшем библиотеки наиболее употребляемых программ.

3. Упрощается процедура загрузки больших программ в ОП, когда требуется ее сегментация.

4. Возникает много естественных контрольных точек для наблюдения за осуществлением хода разработки программ, а в последующем за ходом исполнения программ.

5. Обеспечивается более эффективное тестирование программ, проще осуществляется проектирование и последующая отладка.

**16.09.19**

**Тестирование программного обеспечения - это процесс исследования ПО с целью получения информации о качестве продукта.**

**Виды тестирования по целям:**

Функциональная - направленна на проверку того какие функции программного продукта реализованы и того на сколько верно они реализованы. Состав функционального тестирования: требования, анализ, тест кейсы, ручное тестирование, автоматизированное тестирование, исследовательское тестирование и отчетность.

Не функциональное тестирование - проверяет корректность работы нефункциональных требований. Этот вид тестирования включает в себя тестирование производительности: нагрузочная, стрессовая. Тестирование стабильности, конфигурационная, тестирование пользовательского интерфейса, тестирование удобства использования, тестирование защищенности, инсталяционное тестирование, тестирование совместимости, тестирование надежности, тестирование локализации .

**По степени автоматизации:** ручное и автоматизированное.

**По позитивности сценария** бывают позитивным (проверка программного продукта на соответствие ожидаемому поведению) и негативный (проверяет будет ли программный продукт работать в случае, когда поведение пользователя отличается от ожидаемого.

**По доступу к коду программного продукта** делится на тестирование белого ящика (тестирование программного продукта с доступом к коду) и тестирование черного ящика (тестирование без доступа к коду) и тестирование серого ящика (тестирование, основанное на ограниченном знании внутренней структуры программного продукта.

**По уровню:**

1. Модульное (unit) проверка корректной работы отдельных единиц системы программного продукта.

2. Интеграционное тестирование - проверка взаимодействия между несколькими единицами программного продукта.

3) Системная - проверка работы всей системы на соответствие заявленным требованиям к программному продукту.

**По исполнителю:** альфа тестирование и бета тестирование.

**По формальности:** тестирование по тестам, исследовательское тестирование и свободное тестирование.

**По важности:** дымное тестирование, тестирование критического пути, расширенное тестирование.

**16.09.19**

**Методы проектирования алгоритмов.**

Методы проектирования алгоритмов можно классифицировать по признакам:

1. Степень автоматизации проектных работ. По степени автоматизации проектирования алгоритмов можно выделить: методы традиционного проектирования и методы автоматизированного проектирования.

2. Принятая методология процесса разработки.

Проектирование алгоритмов основывается на различных подходах:

1. Структурное проектирование.

2. Информационное моделирование предметной области и связанных с ней приложений.

3. Объектно-ориентированное проектирование программных продуктов.

В зависимости от объекта структурирования различают:

1. Функционально-ориентированные методы.

2. Методы структурирования данных.

Структурный подход используют:

1. Диаграммы потоков данных.

2. Интегрированную структуру данных предметной области.

3. Диаграммы декомпозиции.

4. Структурные схемы.

Даталогические модели имеют логический и физический уровни. Физический уровень соответствует организации хранения данных в памяти компьютера, а логический уровень реализован в виде:

1. Концептуальной модели БД.

2. Внешних моделей данных.

Обьектно-ориентированный подход основан на:

1. Выделении классов объектов.

2. Установлении характерных свойств объектов и методов их обработки.

3. Создание иерархии классов, наследование свойств объектов и методов их обработки.

Объектно-ориентированный анализ - это анализ предметной области и выделение объектов, определение свойств и методы обработки объектов, установление их взаимосвязей.

Объектно-ориентированное проектирование соединяет процесс объектной декомпозиции и представления с использованием моделей данных, проектируемой системы на логическом и физическом уровнях в статике и динамике.

**18.09.19**

**Современные языки программирования.**

1. JavaScript - прототипно-ориентированный сценарный язык программирования.

2. Java - строго типизированный объектно-ориентированный язык программирования.

3. PHP - является распространенным интерпретируемым языком общего назначения с открытым исходным кодом. Язык для web-разработки.

4. Python - высокоуровневый язык общего назначения, ориентированный на повышение производительности разработчика, читаемости кода и на разработку web-приложений.

5. C# - объектно-ориентированный язык программирования с синтаксисом ближе к C++, имеет статистическую типизацию, поддерживает полиморфизм, перезагрузку операторов, делегаты, атрибуты, события, свойства, обобщенные типы и методы, итераторы, анонимные функции.

6. C++ - компилируемый статистически-типизированный язык общего назначения.

7. Ruby - ориентирован на разработку web-приложений, обладает независимой от ОС реализацией многопоточности, строго динамической типизацией, сборщиком мусора.

8. CSS - формальный язык описания внешнего вида документа, написанного с использованием языка разметки.

9. C - компилируется статистически-типизированный язык программирования общего назначения для разработки низкоуровневых приложений.

10. Shell - скрипты используются для автоматизации обновления ПО.

11. R - язык для статистической обработки данных и работы с графикой, также свободная программная среда вычислений с открытыми исходными кодом.

12. Perl - высокоуровневый интерпретируемый динамический язык общего назначения. Особенности языка - это работа с текстом и с выражениями.

13. Scala - мульти-парадигмальный язык программирования, спроектированный кратким и типобезопасным, для простого и быстрого создания компонентного ПО.

14. Go - компилируемый многопоточный язык, разрабатывался как язык системного программирования.

15. SQL - языке структурированных запросов.

16. Husckell - стандартизированный чистый функциональный язык программирования общего назначения.

17. Swift - открытый мульти-парадигмальный компилируемый язык программирования общего назначения.

18. Matlab - высокоуровневый интерпретируемый язык программирования, основанный на матрицах структуры данных. Программы, написанные на нем, бывают 2х типов: функции и скрипты.

19. VisualBasic.

20. Delphi.

21. Groovy - объектно-ориентированный язык, работающий для платформы java как дополнение.

22. Assembler - низкоуровневый машино-ориентированный язык программирования.

23. D - мульти-парадигмальный язык программирования похожий на C++.

**20.09.19**

**Понятие блох схемы по ГОСТу.**

При описании алгоритмов используются блок схемы. Согласно ГОСТу 19.701-90 под схемой понимается графическое представление определения, анализа или метода решения задачи. Символы, приведенные в государственном стандарте, могут использоваться в следующих типах схем:

1. Схемы данных определяют последовательность обработки данных и их носителей.

2. Схемы программ отображают последовательность операций в программе.

3. Схемы работы системы отображают управление операциями и потоки данных в системе.

4. Схемы взаимодействия программ отображают путь активации программ и их взаимодействия с соответствующими данными.

5. Схемы ресурсов системы отображают конфигурацию блоков данных и обрабатывающих блоков.

Элементы графической нотации:

1. Символы данных указывают на наличие данных, вид носителя или способ ввода/вывода данных.

2. Символы процесса указывают операции, которые следует выполнить над данными.

3. Символы линий указывают потоки данных между процессами или носителями данных, а также потоки управления между процессами.

4. Специальные символы используются для обеспечения написания и чтения схем.

Каждую категорию символов делят на основные и специфические. Основной символ используется в тех случаях, когда точный вид процесса или носителя данных неизвестен или отсутствует необходимость в описании фактического носителя данных. Специфический символ используется в тех случаях, когда известен точный вид процесса или носителя данных и это необходимо отразить на схеме.

**Правила и рекомендации построения схем:**

1. Допускается зеркально отображать символы и поворачивать их вокруг оси.

2. Большинство символов запускают задание внутри них текстовых пояснений.

3. Количество пересечений линий следует минимизировать.

4. Если две или более линии объединены в одну, то место объединения должно быть смещено.

5. Несколько выходов из символа решения сделает показывать одним из способов:

1) Несколькими линиями от данного стала к другим символам.

2) Одной линией от данного символа, которая потом разветвляется.

6. Вместо одного символа с соответствующим текстом могут быть использованы несколько символов с перекрытием изображения, каждый из которых может содержать дополнительный текст.

7. Если направление стрелки не указано, то направление потока считается сверху вниз, или слева направо.

**23.09**

**Особенности анализа и проектирования крупных систем.**

**Особенности:**

1. Сложность описания требует тщательного моделирования и анализа данных и процессов.

2. Наличие совокупности тесно взаимодействующих элементов.

3. Отсутствие прямых аналогов, ограничивающая возможность использования каких-либо типовых проектных решений и прикладных систем.

4. Необходимость интеграции существующих и вновь разрабатываемых подсистем

5. Функционирование в неоднородной среде на разных операционных платформах.

6. Разобщенность и разнородность отдельных групп разработчиков по уровню квалификации и сложившимся традициям использования техзадания или иных инструментальных средств.

7. Существенная временная протяженность проекта, обусловленная ограниченными возможностями коллектива разработчиков, масштабами организации заказчика, различной степени готовности отдельных ее подразделений к внедрению информационных систем.

8. Изменение или уточнение потребностей пользователей в процессе разработки и эксплуатации системы.

9. Основные требования на разработку информационной системы документально оформляются в виде календарного плана или технического оборудования. Документы содержащие требования на разработку системы.

Календарный план регламентирует состав, сроки и финансирование работ. Тех задание - основные требования к системе.

**Техзадание может включать:**

1. Расчет ожидаемой эффективности системы.

2. Оценка научно-технического уровня системы.

3. Перечень основных входных и выходных форм.

**Техзадание содержит следующие разделы:**

1. Общие сведения: наименование системы, наименование предприятия разработчика и заказчика с их реквизитами, перечень документов на основании которых создается система, плановые сроки начала и окончания работы

2 Назначения и цели создания системы

3. Характеристика объектов автоматизации

4. Требования:

**1) К системе в целом:** требования к функционированию и структуре, к численности и квалификации персонала, показатели назначения, к надежности, к безопасности, к эргономике и тех эстетике, к транспортабельности, к эксплуатации, к техническому обслуживанию, к ремонту и хранению компонентов системы, требование защиты информации от несанкционированного доступа, требования к сохранности информации при аварии, требование к защите от внешних воздействий, к патентной чистоте, требование по стандартизации и унификации.

**2) Требования к функциям:** перечень функций, задач или их комплексов, подлежащих автоматизации, временной регламент реализации каждой функции, требования к качеству реализации каждой функции, к форме выходной информации, характеристики необходимой точности и времени выполнения, требования к одновременности выполнения группы функций, достоверности выдачи результатов, перечень и критерии отказов для каждой функции по которой задаются требования по надежности.

**3) Требования к обеспечению:**

I. Математическому обеспечению – совокупности математических методов, моделей и алгоритмов, применяемых в информационной системе.

II. Информационному обеспечению – совокупности форм документов, классификаторов, нормативной базы и реализованных решений по объемам, размещению и формам существования информации.

III. Лингвистическому обеспечению – совокупности правил применения в системе языков программирования, языков взаимодействия пользователей и технических средств системы, а также совокупности требований к кодированию и декодированию данных и способам организации диалога.

IV. Программному обеспечению – совокупности программ и документов, предназначенных для отладки, функционирования и проверки работоспособности системы.

V. Техническому обеспечению – совокупности всех технических средств, используемых при функционировании системы.

VI. Метрологическому обеспечению – совокупности норм, правил и методик выполнения измерений.

VII. Организационному обеспечению – совокупности документов, устанавливающих организационную структуру, права и обязанности пользователей и эксплуатационного персонала системы.

VIII. Методическому обеспечению – совокупности документов, описывающих технологию функционирования системы, методы выбора и применения пользователями технологических приемов для получения конкретных результатов.

5. Состав и содержание работ по созданию системы.

6. Порядок контроля и приемки системы.

7. Требования к составу и содержанию работ по подготовке объектов автоматизации к выводу системы в действие.

8. Требование к документированию.

9. Источники разработки, документы и информационные материалы, на основании которых разрабатывалось техническое задание, и которые должны быть использованы при создании системы.

**Основные принципы проектирования.**

Процесс перехода от первичного описания системы в виде технического задания к ее описанию в виде набора стандартных документов, достаточных для создания системы, называется **проектированием.**

Принципы проектирования:

1. Принцип декомпозиции – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач легких для понимания и решения.

2. Принцип иерархического упорядочения.

3. Принцип концептуальной общности.

4. Принцип абстрагирования.

5. Принцип формализации заключается в необходимости строгого методического подхода к решению проблемы и описании системы на формальном языке, пригодном для ее анализа, проектирования и разработки, а также автоматизированной генерации кодов БД.

6. Принцип унификации предписывает унифицированное представление и обозначение одного и того же элемента или однотипных элементов в разных моделях.

7. Принцип логической независимости.

8. Принцип многомодельности.

9. Принцип непротиворечивости.

10. Принцип информационной закрытости.

11. Принцип полиморфизма.

**Классификация моделей информационной системы.**

**По строгости описания:** формальные и неформальные.

**По степени физической реализации:** логические и физические.

**По степени отображения динамики, происходящих процессов:** статические и динамические. Статические описывают состав и структуру системы, а динамические описывают поведение системы или ее отдельных элементов.

**По отображаемому аспекту:** функциональные (описывают функции системы и возможные варианты ее использования), информационные (описывают состав и структуру данных), поведенческие (описывают состояние системы или ее отдельных элементов и переходы между ними, взаимодействие элементов, алгоритмы обработки информации), компонентные (описывают состав и структуру программных и аппаратных средств), смешанные (характеризуют сразу несколько аспектов системы).

**Основы программирования в C#.**

С# - объектно-ориентированный язык программирования. Язык имеет синтаксическую типизацию, поддерживает полиморфизм, перезагрузку операторов, делегаты, атрибуты, события, свойства, обобщенные типы и методы, итераторы, анонимные функции с поддержкой замыканий, исключения, комментарии в формате XML.

**1. Инструкции.**

Базовым строительным блоком программы являются инструкции. Инструкция представляет некоторое действие. В конце каждой инструкции ставиться ";". Например: Console.writeline ("..."); Данная строка представляет вызов метода, который выводит на консоль строку. Набор инструкций, объединенный в блок, заключается в фигурной скобке. Одни блоки могут содержать другие блоки.

**2. Метод main.**

Метод main является точкой входа в программу на языке С#. По умолчанию метод main размещается в классе programm. Метод main является обязательной частью консольного приложения.

**3. Регистрозависимость**.

**4. Комментарии.**

Однострочный комментарий размещается на одной строке после //. Многострочный комментарий заключается /\*...\*/

**5. Переменные.**

Определение переменной: тип имя переменной;

В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет требованиям:

1) имя может содержать любые цифры, буквы и символ "\_", при этом первый символ в имени должен быть буквой или "\_".

2) в имени не должно быть знаков пунктуации и пробелов.

3) имя не должно быть ключевым словом языка С#.

String name;

String Name; (2 разные)

name="tom";

Отличительной чертой переменных является то, что в программе можно многократно менять их значение.

**6. Логические литералы.**

Литералы представляют неизменяемые значения. Литералы можно передавать переменным в качестве значения. Литералы бывают логическими, целочисленными, вещественными, символьными (первый символ в одинарных кавычках) и строчными и отдельный литерал "null".

**Управляющие последовательности:**

1. \n - перевод строки.

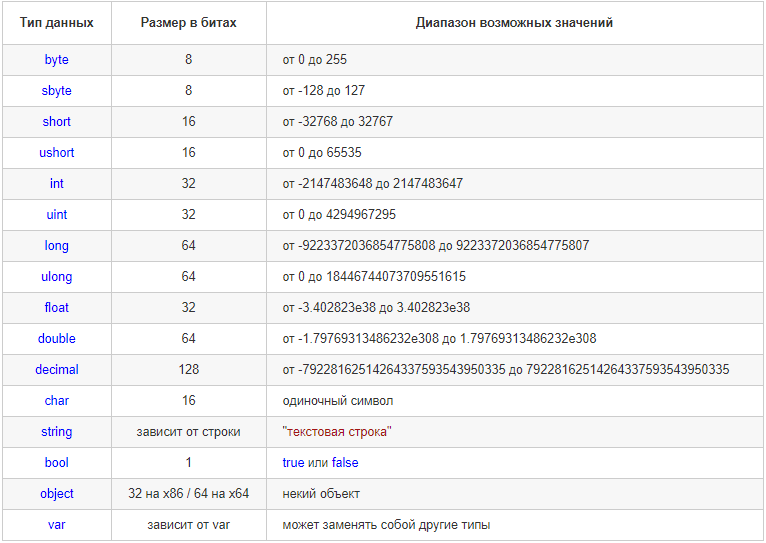
2. \t – табуляция.

3. \ обратный слэш.

Строковые литералы заключаются в " ("ООО \"Ваз\"")

В строках можно использовать управляющие последовательности.

**7. Типы данных:**



09.10 МАССИВЫ

**Общий взгляд**

**Массивом** называют упорядоченную совокупность элементов одного типа. Каждый элемент массива имеет индексы, определяющие порядок элементов. Число индексов характеризует **размерность массива**. В языке C#, как и во многих других языках, индексы задаются целочисленным типом. Каждый индекс изменяется в некотором диапазоне [a,b]. Диапазон [a,b] называется граничной парой, a – нижней границей, b – верхней границей индекса.

При объявлении массива границы задаются выражениями. Если все границы заданы константными выражениями, то число элементов массива известно в момент его объявления и ему может быть выделена память еще на этапе трансляции. Такие массивы называются статическими. Если же выражения, задающие границы, зависят от переменных, то такие массивы называются динамическими, поскольку память им может быть отведена только динамически в процессе выполнения программы, когда становятся известными значения соответствующих переменных. Массиву, как правило, выделяется непрерывная область памяти. Массивы в языке C# являются динамическими.

В языке C#, соблюдая преемственность, сохранены одномерные массивы и массивы массивов. В дополнение к ним в язык добавлены многомерные массивы.

**Объявление массивов**

**Одномерные массивы:**

Общая структура объявления: [<атрибуты>] [<модификаторы>] <тип> <объявители>;

Объявление одномерного массива: <тип>[] <объявители>;

Квадратные скобки приписаны не к имени переменной, а к типу. Они являются неотъемлемой частью определения типа, так что запись T[] следует понимать как тип, задающий **одномерный массив с элементами типа T**.

Как и в случае объявления простых переменных, каждый объявитель может быть именем с отложенной инициализацией или именем с инициализацией. Нужно понимать, что при объявлении с отложенной инициализацией сам массив не формируется, а создается только ссылка на массив и пока массив не будет реально создан и его элементы инициализированы, использовать его в вычислениях нельзя.

Отложенная инициализация: int[] a, b, c;

Имя с инициализацией:

Константный массив: double[] x = {5.5, 6.6, 7.7};

При помощи конструктора: int[] d = new int[5];

public void TestDeclaration()

{

//объявляются три одномерных массива A,B,C

int[] A = new int[5], B= new int[5], C= new int[5];

//создание массивов статическим методом класса Arrs

Arrs.CreateOneDimAr(A);

Arrs.CreateOneDimAr(B);

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

C[i] = A[i] + B[i];

//объявление массива с явной инициализацией

int[] x ={5,5,6,6,7,7};

//объявление массивов с отложенной инициализацией

int[] u,v;

u = new int[3];

for(int i=0; i<3; i++) u[i] =i+1;

// v = {1,2,3}; //присваивание константного массива недопустимо

v = new int[4];

v = u; //допустимое присваивание

Arrs.PrintAr1("A", A); Arrs.PrintAr1("B", B);

Arrs.PrintAr1("C", C); Arrs.PrintAr1("X", x);

Arrs.PrintAr1("U", u); Arrs.PrintAr1("V", v);

}

На что следует обратить внимание, анализируя этот текст:

• В процедуре показаны разные способы объявления массивов. Вначале объявляются одномерные массивы A, B и C, создаваемые конструктором. Значения элементов этих трех массивов имеют один и тот же тип int. То, что они имеют одинаковое число элементов, произошло по воле программиста, а не диктовалось требованиями языка. Заметьте, что после такого объявления с инициализацией конструктором, все элементы имеют значение, в данном случае – ноль, и могут участвовать в вычислениях.

• Массив x объявлен с явной инициализацией. Число и значения его элементов определяется константным массивом.

• Массивы u и v объявлены с отложенной инициализацией. В последующих операторах массив u инициализируется в объектном стиле, его элементы получают в цикле значения.

• Обратите внимание на закомментированный оператор присваивания. В отличие от инициализации, использовать константный массив в правой части оператора присваивания недопустимо. Эта попытка приводит к ошибке, поскольку v — это ссылка, которой можно присвоить ссылку, но нельзя присвоить константный массив. Ссылку присвоить можно.

Что происходит в операторе присваивания v = u.? Это корректное ссылочное присваивание: хотя u и v имеют разное число элементов, но они являются объектами одного класса. В результате присваивания память, отведенная массиву v, освободится, ей займется теперь сборщик мусора. Обе ссылки u и v будут теперь указывать на один и тот же массив, так что изменение элемента одного массива немедленно отражается на другом массиве.

• Далее определяется двумерный массив w и делается попытка выполнить оператор присваивания v = w. Это ссылочное присваивание некорректно, поскольку объекты w и v разных классов и для них не выполняется требуемое для присваивания согласование по типу.

• Для поддержки работы с массивами создан специальный класс Arrs, статические методы которого выполняют различные операции над массивами. В частности, в примере использованы два метода этого класса, один из которых заполняет массив случайными числами, второй – выводит массив на печать. Вот текст первого из этих методов:

public static void CreateOneDimAr(int[] A)

{

for(int i = 0; i<A.GetLength(0);i++)

A[i] = rnd.Next(1,100);

}//CreateOneDimAr

Здесь rnd – это статическое поле класса Arrs, объявленное следующим образом:

private static Random rnd = new Random();

Процедура печати массива с именем name выглядит так:

public static void PrintAr1(string name,int[] A)

{

Console.WriteLine(name);

for(int i = 0; i<A.GetLength(0);i++)

Console.Write("\t" + name + "[{0}]={1}", i, A[i]);

Console.WriteLine();

}//PrintAr1

**Динамические массивы:**

Чисто синтаксически нет существенной разницы в объявлении статических и динамических массивов. Выражение, задающее границу изменения индексов, в динамическом случае содержит переменные. Единственное требование – значения переменных должны быть определены в момент объявления. Это ограничение в C# выполняется, поскольку C# контролирует инициализацию переменных.

Приведу пример, в котором описана работа с динамическим массивом:

public void TestDynAr()

{

//объявление динамического массива A1

Console.WriteLine("Введите число элементов массива A1");

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

int[] A1 = new int[size];

Arrs.CreateOneDimAr(A1);

Arrs.PrintAr1("A1",A1);

}//TestDynAr

**Многомерные массивы:**

**Одномерные массивы** — это частный случай многомерных. Можно говорить и по-другому: многомерные массивы являются естественным обобщением одномерных. Одномерные массивы позволяют задавать такие математические структуры, как векторы, двумерные – матрицы, трехмерные – кубы данных, массивы большей размерности — многомерные кубы данных.

Объявление многомерного массива: <тип>[, … ,] <объявители>;

Число запятых, увеличенное на единицу, и задает размерность массива. Что касается объявителей, то все, что сказано для одномерных массивов, справедливо и для многомерных.

**Массивы массивов (изрезанные массивы (jagged arrays)):**

Это одномерный массив, элементами которого являются массивы, элементы которых тоже могут быть массивами и т.д.

Если при объявлении типа многомерных массивов для указания размерности использовались запятые, то для изрезанных массивов применяется более ясная символика – совокупности пар квадратных скобок; например, int[][] задает массив, элементы которого — одномерные массивы элементов типа int.

Сложнее с созданием самих массивов и их инициализацией. Здесь нельзя вызвать конструктор new int[3][5], поскольку он не задает изрезанный массив. Фактически нужно вызывать конструктор для каждого массива на самом нижнем уровне. В этом и состоит сложность объявления таких массивов. Начну с формального примера:

//массив массивов - формальный пример

//объявление и инициализация

int[][] jagger = new int[3][]

{

new int[] {5,7,9,11},

new int[] {2,8},

new int[] {6,12,4}

};

**Процедуры и массивы**

В наших примерах массивы неоднократно передавались процедурам в качестве входных аргументов и возвращались в качестве результатов. Остается подчеркнуть только некоторые детали:

• В процедуру достаточно передавать только сам объект – массив. Все его характеристики (размерность, границы) можно определить, используя свойства и методы этого объекта.

• Когда массив является выходным аргументом процедуры, как аргумент C в процедуре MultMatr, выходной аргумент совсем не обязательно снабжать ключевым словом ref или out (хотя и допустимо). Передача аргумента по значению в таких ситуациях так же хороша, как и передача по ссылке. В результате вычислений меняется сам массив в динамической памяти, а ссылка на него остается постоянной. Процедура и ее вызов без ключевых слов выглядит проще, поэтому обычно они опускаются. Заметьте, в процедуре GetSizes, где определялись границы массива, ключевое слово out, сопровождающее аргументы, совершенно необходимо.

• Функция может возвращать массив в качестве результата.

**Алгоритмы и задачи**

**Массив** – это упорядоченная последовательность элементов одного типа. Порядок элементов задается с помощью индексов.

**Ввод – вывод массивов:**

Возможны три основных способа:

• Вычисление значений в программе;

• значения вводит пользователь;

• связывание с источником данных.

**МЕТОДЫ**

Метод - именованный блок кода, который выполняет некоторые действия. Методы содержат набор операторов, которые выполняют определенные действия.

Общее определение методов выглядит следующим образом:

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

{

// тело метода

}

Модификаторы и параметры необязательны. Консольная программа на языке C# должна содержать как минимум один метод - метод Main, который является точкой входа в приложение:

static void Main(string[] args)

{

//тело метода

}

Слово static является модификатором, слово void указывает на то, что метод ничего не возвращает.

Вызов методов.

Чтобы использовать методы в программе, нам надо вызвать их в методе Main. Для вызова метода указывается его имя, после которого в скобках идут значения для его параметров (если метод принимает параметры):

название\_метода (значения\_для\_параметров\_метода);

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

Возвращение значения.

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

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

{

// тело метода;

return возвращаемое значение;

}

После оператора return также можно указывать сложные выражения, которые возвращают определенный результат. При этом методы, которые в качестве возвращаемого типа имеют любой тип, отличный от void, обязательно должны использовать оператор return для возвращения значения. Также между возвращаемым типом метода и возвращаемым значением после оператора return должно быть соответствие. Результат методов, который возвращают значение, мы можем присвоить переменным или использовать иным образом в программе.

Выход из метода.

Оператор return не только возвращает значение, но и производит выход из метода. Поэтому он должен определяться после остальных инструкций. Можно использовать оператор return и в методах с типам void. В этом случае после оператора return не ставится никакого возвращаемого значения (ведь метод ничего не возвращает). Типичная ситуация - в зависимости от определенных условий произвести выход из метода.

Сокращенная запись методов.

Если метод в качестве тела определяет только одну инструкцию, то мы можем сократить определение метода. Например, допустим у нас есть метод:

static void SayHello()

{

Console.WriteLine("Hello");

}

Мы можем его сократить следующим образом:

static void SayHello() => Console.WriteLine("Hello");

То есть после списка параметров ставится знак «равно» и «больше чем», после которого идет выполняемая инструкция.

Подобным образом мы можем сокращать методы, которые возвращают значение:

static string GetHello()

{

return "hello";

}

Анлогичен следующему методу:

static string GetHello() => "hello";

Параметры методов

Параметры позволяют передать в метод некоторые входные данные. Например, определеим метод, который складывает два числа:

static int Sum(int x, int y)

{

return x + y;

}

Значения, которые передаются параметрам, еще называются аргументами.

Иногда можно встретить такие определения как формальные параметры и фактические параметры. Формальные параметры - это собственно параметры метода, а фактические параметры - значения, которые передаются формальным параметрам. То есть фактические параметры - это и есть аргументы метода.

Передаваемые параметру значения могут представлять значения переменных или результат работы сложных выражений, которые возвращают некоторое значение.

Если параметрами метода передаются значения переменных, которые представляют базовые примитивные типы (за исключением типа object), 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# также позволяет использовать необязательные параметры. Для таких параметров нам необходимо объявить значение по умолчанию. Также следует учитывать, что после необязательных параметров все последующие параметры также должны быть необязательными:

static int OptionalParam(int x, int y, int z=5, int s=4)

{

return x + y + z + s;

}

Так как последние два параметра объявлены как необязательные, то мы можем один из них или оба опустить:

static void Main(string[] args)

{

OptionalParam(2, 3);

OptionalParam(2,3,10);

Console.ReadKey();

}

Именованные параметры.

Также можно использовать именованные параметры.

static int OptionalParam(int x, int y, int z=5, int s=4)

{

return x + y + z + s;

}

static void Main(string[] args)

{

OptionalParam(x:2, y:3);

//Необязательный параметр z использует значение по умолча- нию.

OptionalParam(y:2, x:3, s:10);

Console.ReadKey();

}

Передача параметров по ссылке и значению. Выходные и входные параметры.

Существует два способа передачи параметров в метод в языке C#: по значению и по ссылке.

**Передача параметров по значению:**

Sum(10, 15); // параметры передаются по значению

**Передача параметров по ссылке и модификатор ref:**

int x = 10;

int y = 15;

Addition(ref x, y); // вызов метода

Модификатор ref указывается, как при объявлении метода, так и при его вызове в методе Main.

static void Addition(ref int x, int y)

{

x += y;

}

**Сравнение передачи по значению и по ссылке:**

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

**Выходные параметры. Модификатор out:**

Чтобы сделать параметр выходным, перед ним ставится модификатор out:

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

{

a = x + y;

}

Здесь результат возвращается не через оператор return, а через выходной параметр.

Причем, как и в случае с ref ключевое слово out используется как при определении метода, так и при его вызове.

Также обратите внимание, что методы, использующие такие параметры, обязательно должны присваивать им определенное значение. То есть следующий код будет недопустим, так как в нем для out-параметра не указано никакого значения:

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

{

Console.WriteLine(x+y);

}

Прелесть использования подобных параметров состоит в том, что по сути мы можем вернуть из метода не один вариант, а несколько. Например:

static void Main(string[] args)

{

int x = 10;

int area;

int perimetr;

GetData(x, 15, out area, out perimetr);

Console.WriteLine("Площадь : " + area);

Console.WriteLine("Периметр : " + perimetr);

Console.ReadKey();

}

static void GetData(int x, int y, out int area, out int perim)

{

area= x \* y;

perim= (x + y)\*2;

}

Rак и в случае с ключевым словом ref, ключевое слово out применяется для передачи аргументов по ссылке. Однако в отличие от ref для переменных, которые передаются с ключевым словам out, не требуется инициализация. И кроме того, вызываемый метод должен обязательно присвоить им значение.

Стоит отметить, что начиная с версии C# 7.0 можно определять переменные в непосредственно при вызове метода.

**Входные параметры. Модификатор in:**

Кроме выходных параметров с модификатором out метод может использовать входные параметры с модификатором in. Модификатор in указывает, что через данный параметр будет передаваться в метод по ссылке, однако внутри метода его значение параметра нельзя будет изменить. Например, возьмем следующий метод:

static void GetData(in int x, int y, out int area, out int perim)

{

// x = x + 10; нельзя изменить значение параметра x

y = y + 10;

area = x \* y;

perim = (x + y) \* 2;

}

В данном случае через параметры x и y в метод передаются значения, но в самом методе можно изменить только значение параметра y, так как параметр x указан с модификатором in.

КЛАССЫ

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описанием объекта является класс, а объект представляет экземпляр этого класса.

По умолчанию проект консольного приложения уже по умолчанию содержит один класс Program, с которого и начинается выполнение программы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова сlass:

class Person

{

…

}

Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Поскольку класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип.

Конструкторы.

Кроме обычных методов в классах используются также и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Конструкторы выполняют инициализацию объекта. Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Создадим один объект класса Person:

class Program

{

static void Main(string[] args)

{

Person tom = new Person();

tom.GetInfo(); // Имя: Возраст: 0

tom.name = "Tom";

tom.age = 34;

tom.GetInfo(); // Имя: Tom Возраст: 34

Console.Read();

}

}

Для создания объекта Person используется выражение new Person(). Оператор new выделяет память для объекта Person. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта Person. А переменная tom получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа string и классов - это значение null (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта Person через переменную tom и установить или получить их значения, например, tom.name = "Tom";

Создание конструкторов.

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить свои конструкторы:

class Person

{

public string name;

public int age;

public Person() { name = "Неизвестно"; age = 18; } // 1 кон структор

public Person(string n) { name = n; age = 18; } // 2 конструк тор

public Person(string n, int a) { name = n; age = a; } // 3 конструк тор

public void GetInfo()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

static void Main(string[] args)

{

Person tom = new Person(); // вызов 1-ого конструктора без параметров

Person bob = new Person("Bob"); //вызов 2-ого конструктора с одним параметром

Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами

bob.GetInfo(); // Имя: Bob Возраст: 18

tom.GetInfo(); // Имя: Неизвестно Возраст: 18

sam.GetInfo(); // Имя: Sam Возраст: 25

Console.ReadKey();

}

Если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

Ключевое слово this.

Ключевое слово this представляет ссылку на текущий экземпляр класса. В каких ситуациях оно нам может пригодиться? В примере выше определены три конструктора. Все три конструктора выполняют однотипные действия - устанавливают значения полей name и age. Но этих повторяющихся действий могло быть больше. И мы можем не дублировать функциональность конструкторов, а просто обращаться из одного конструктора к другому через ключевое слово this, передавая нужные значения для параметров:

class Person

{

public string name;

public int age;

public Person() : this("Неизвестно")

{

}

public Person(string name) : this(name, 18)

{

}

public Person(string name, int age)

{

this.name = name;

his.age = age;

}

public void GetInfo()

{

Console.WriteLine($"Имя: {name} Возраст: {age}");

}

}

В данном случае первый конструктор вызывает второй, а второй конструктор вызывает третий. По количеству и типу параметров компилятор узнает, какой именно конструктор вызывается. Например, во втором конструкторе:

public Person(string name) : this(name, 18)

{

}

идет обращение к третьему конструктору, которому передаются два значения. Причем в начале будет выполняться именно третий конструктор, и только потом код второго конструктора.

Также стоит отметить, что в третьем конструкторе параметры называются также, как и поля класса.

public Person(string name, int age)

{

this.name = name;

this.age = age;

}

И чтобы разграничить параметры и поля класса, к полям класса обращение идет через ключевое слово this. Так, в выражении this.name = name; первая часть this.name означает, что name - это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы необязательно. Также через ключевое слово this можно обращаться к любому полю или методу.

Инициализаторы объектов.

Для инициализации объектов классов можно применять инициализаторы. Инициализаторы представляют передачу в фигурных скобках значений доступным полям и свойствам объекта:

Person tom = new Person { name = "Tom", age=31 };

tom.GetInfo(); // Имя: Tom Возраст: 31

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

При использовании инициализаторов следует учитывать следующие моменты:

• С помощью инициализатора мы можем установить значения только доступных из внешнего кода полей и свойств объекта. Например, в примере выше поля name и age имеют модификатор доступа public, поэтому они доступны из любой части программы.

• Инициализатор выполняется после конструктора, поэтому если и в конструкторе, и в инициализаторе устанавливаются значения одних и тех же полей и свойств, то значения, устанавливаемые в конструкторе, заменяются значениями из инициализатора.

Свойства и инкапсуляция.

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.

Стандартное описание свойства имеет следующий синтаксис:

[модификатор\_доступа] возвращаемый\_тип произвольное\_название

{

// код свойства

}

Например:

class Person

{

private string name;

public string Name

{

get

{

return name;

}

set

{

name = value;

}

}

}

Здесь у нас есть закрытое поле name и есть общедоступное свойство Name. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной name. Стандартное определение свойства содержит блоки get и set. В блоке get мы возвращаем значение поля, а в блоке set устанавливаем. Параметр value представляет передаваемое значение.

Мы можем использовать данное свойство следующим образом:

Person p = new Person();

// Устанавливаем свойство - срабатывает блок Set

// значение "Tom" и есть передаваемое в свойство value

p.Name = "Tom";

// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get

string personName = p.Name;

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют только блок get, то такое свойство доступно только для чтеня - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

Модификаторы доступа

Мы можем применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам - либо get, либо set:

class Person

{

private string name;

public string Name

{

get

{

return name;

}

private set

{

name = value;

}

}

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

Теперь закрытый блок set мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

Person p = new Person("Tom", 24);

// Ошибка - set объявлен с модификатором private

//p.Name = "John";

Console.WriteLine(p.Name);

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

• Модификатор для блока set или get можно установить, если свойство имеет оба блока (и set, и get)

• Только один блок set или get может иметь модификатор доступа, но не оба сразу

• Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы protected internal, internal, protected, private.

Инкапсуляция.

Через свойства устанавливается доступ к приватным переменным класса. Подобное сокрытие состояния класса от вмешательства извне представляет механизм инкапсуляции, который представляет одну из ключевых концепций объектно-ориентированного программирования. Применение модификаторов доступа типа private защищает переменную от внешнего доступа. Для управления доступом во многих языках программирования используются специальные методы, геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Автоматические свойства.

Свойства управляют доступом к полям класса. Однако что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в фреймворк .NET были добавлены автоматические свойства. Они имеют сокращенное объявление:

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции. В любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

class Person

{

public string Name { get; set; } = "Tom";

public int Age { get; set; } = 23;

}

class Program

{

static void Main(string[] args)

{

Person person = new Person();

Console.WriteLine(person.Name); // Tom

Console.WriteLine(person.Age); // 23

Console.Read();

}

}

И если мы не укажем для объекта Person значения свойств Name и Age, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа:

class Person

{

public string Name { private set; get;}

public Person(string n)

{

Name = n;

}

}

Мы можем убрать блок set и сделать автосвойство доступным только для чтения. В этом случае для хранения значения этого свойства для него неявно будет создаваться поле с модификатором readonly, поэтому следует учитывать, что подобные get-свойства можно установить либо из конструктора класса, как в примере выше, либо при инициализации свойства:

class Person

{

public string Name { get;} = "Tom"

}

Сокращенная запись свойств.

Как и методы, мы можем сокращать свойства. Например:

class Person

{

private string name;

// эквивалентно public string Name { get { return name; } }

public string Name => name;

}