|  |  |  |
| --- | --- | --- |
| 1. | | Основные элементы интерфейса Visual Studio 2019 и их назначение. Понятие конструктора формы и работа в этом режиме. Примеры и/или демонстрация. |
| 1. - главное меню среды разработки. Пункт меню «Файл» служит для сохранения отдельных файлов проекта и загрузки существующих проектов. Пункт меню «Правка» содержит различные полезные инструменты для редактирования кода. Пункт меню «Вид» служит для открытия или закрытия дополнительный стыкуемых или диалоговых окон для выполнения различных действий. Пункт меню «Проект» служит для добавления к проекту дополнительных файлов и ресурсов, а также позволяет настроить то, как будет «собираться» приложение. Пункты меню «Сборка» и «Отладка» позволяют «собрать» приложение (т.е. сгенерировать объектный код для его выполнения) и запустить его на выполнение. 2. - панель инструментов. На ней находятся кнопки для открытия файлов/проектов, сохранения файлов/проектов, настройки режима сборки, а также кнопки для запуска и остановки текущего приложения. 3. - вкладки открытых файлов.   «MainWindow.xaml» - это автоматический сгенерированный файл, в котором содержится информация о внешнем виде вашего приложения (расположение и внешний вид кнопок, текстовых полей и т.д.). В области (5) отображается его содержимое, в данном случае графическое представление (форма) будущего приложения. Под областью (5) также отображается содержимое этого файла в текстовом формате. **Сам режим, который позволяет графически проектировать вид будущего приложения, называется «конструктор».** Во второй вкладке «MainWindow.xaml.cs» представлен код программы на языке C#, реализующий логику работы программы. Сам режим отображения и редактирования кода называется «код».   1. - «Обозреватель решений». В этом стыкуемом окне отображается структура проекта/решения. Жирным шрифтом выделен текущий проект (проектов может быть открыто несколько). 2. - панель со стыкуемыми окнами, где расположены «Панель элементов» и «Структура документа». В окне «Панель элементов» представлены все графические элементы, доступные для размещения на форме. Это различные кнопки, текстовые поля, ползунки, полосы прогресса, вкладки, полосы прокрутки, полотно для рисования и т.д. С этими графическими элементами можно создавать приложения любой сложности для любых целей. В окне «Структура документа» представлена текущая структура графического интерфейса приложения. Это окно позволяет удобно находить и выделять элементы на форме.   Как говорилось ранее: Конструктор формы - режим графического проектирования приложения, с помощью элементов используемой нами библиотеки. | | |
| 2. | Основные элементы графического интерфейса из библиотеки WPF. Их назначение и работа с ними. Примеры. | |
| Основные элементы графического интерфейса библиотеки WPF:   * Button - представляет собой элемент, который будет реагировать на событие “click”. То есть в коде прописывается то или иное действие, которое будет вызываться нажатием на элемент Button в приложении. * Canvas - (холст) - представляет собой поле для графического изображения наборов точек в приложении, координаты которых присваиваются в коде. * Grid - это наиболее мощный контейнер компоновки в WPF. Контейнер Grid является идеальным инструментом для разбиения окна на меньшие области, которыми можно управлять с помощью других панелей. * Label - служит текстовым полем для обозначения чего-либо в приложении, не может изменяться пользователем при работе с приложением. * TextBox - текстовое поле для ввода или вывода данных. Например: нам нужно задать значение переменных с клавиатуры при работе с приложением, мы создаем TextBox, в который при запуске приложения можно будет ввести данные, которые затем будут преобразовываться кодом. | | |
| 3. | Понятие события и обработчика события. В чѐм их различия и предназначение. Примеры. | |
| Событие - все, что пользователь может сделать с интерфейсом (щелчок по кнопке, нажатие на клавиши клавиатуры и т.д..)  События генерируются (происходят), когда пользователь взаимодействует с элементами графического интерфейса (окном, текстовым полем, кнопкой и т.д.). События генерируются разного типа в ответ на различные действия пользователя. Одни события генерируются при перемещении курсора мыши в пределах элемента интерфейса, другие при щелчке на элементе интерфейса, третьи при нажатии на клавиши клавиатуры и т.д…  Обработчик событий - это именно то, что выполняется при вызове событий.  Пример: у нас есть кнопка (элемент Button) “Тестовые\_данные”, мы создаем событие и обработчик события, для ее использования: затем в обработчик событий записывается то, что должно произойти при нажатии на “Тестовые\_данные”. | | |
| 4. | Основные типы данных из пространства имѐн System и их сокращѐнные имена. Их назначение. Синтаксические конструкции объявления переменных. Примеры. | |
| Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.  **bool**: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean   1. bool alive = true; 2. bool isDead = false;   **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:   1. int a = 10; 2. int b = 0b101; // бинарная форма b =5 3. int c = 0xFF; // шестнадцатеричная форма c = 255   **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single  **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double  **string**: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.   1. string hello = "Hello"; 2. string word = "world";   **object**: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.   1. object a = 22; 2. object b = 3.14; 3. object c = "hello code"; | | |
| 5. | Основные арифметические и логические операции. Преобразования данных из одного типа в другой. Явные и неявные преобразования типов. Примеры. | |
| Основные арифметические операции:    Основные логические операции:    Язык С# относится к строго типизированным языкам программирования. Это значит, что данные могут преобразовываться из одного типа в другой неявно, если при этом не теряется точность, либо в противном случае необходимо явно указывать преобразование типа.  Примеры:    Тут в третьей строке b сначала было преобразовано к типу int, чтобы можно было сложить два числа b и i, т.е. произошло неявное преобразование типа. А далее результат (тип int) не смог быть неявно преобразован в тип byte, т.к. здесь может произойти потеря значения.  Если потеря значения (точности) допустимо между числовыми типами, то нужно использовать явное приведение типов. Например: | | |
| 6. | Форматирование и работа с текстом. Управляющие символы в тексте. Итерационные строки (с символом $). Буквальные (дословные) строки (с символом @). Примеры. | |
| **Форматирование и работа с текстом**  Для работы с текстом используется тип “string”.  Синтаксис:    Где “s” - название строки, а в кавычки записывается ее содержимое.  Основные управляющие символы:   1. \n – переход на новую строку; 2. \t – горизонтальная табуляция; 3. \\ – позволяет вставить в строку символ «\»; 4. \" – позволяет вставить в строку символ «"».   Для преобразования какого-либо логического типа в строку, используется метод “ToString”. Пример:    Для преобразования строки, в которой содержится корректное значение необходимого типа, используется конструкция:    Либо:    Также для строк существует операция объединения с помощью “+”. Синтаксис:    Конструкция с «$» называется интерполяцией строки. Это конструкция говорит, что дальше идѐт строка, в которой на местах с фигурными скобками нужно вставить соответствующие значения переменных. Фигурные скобки называют элементами форматирования.  Пример:    Символ “@” создает условие, при котором чтение последующей строки читается посимвольно и дословно. Например, для того, чтобы не приходилось писать “\\” вместо “\” для экранирования символа, в пути к файлу перед ним можно добавить “@”, тогда “\” будет читаться именно как “\”, а программа не будет на него “ругаться”. Пример:    Про @ консультация -01:04:00. | | |
| 7. | Понятия: область видимости переменной, блок кода. Синтаксис управляющих конструкций if и switch. Их назначение. Примеры. | |
| (Все примеры были взяты и РТ4. Если подумать, можно сделать нечто похожее) | | |
| **Область видимости переменной, блок кода**  Каждая переменная в программе имеет свою область видимости. Под областью видимости понимается то место (область) в программном коде, где вы можете использовать в выражениях соответствующую переменную. Область видимости любой переменной начинается с момента еѐ объявления и заканчивается в конце блока, в котором она была объявлена. Блоком называется содержимое между открывающей «{» и закрывающей «}» фигурной скобкой. Пример:    **If, switch**  Оператор if применяется тогда, когда необходимо выполнить ту или иную последовательность действий, в зависимости от некоторых условий (условия). Синтаксис оператора if следующий:    В угловых скобках записывается то, что должно быть вместо них на этом месте. Если оказывается истинным, то выполняется. Иначе в первом случае ничего не выполняется и выполнение кода продолжается со следующего оператора, идущего после if. Во втором случае выполняется. Под составным оператором понимается блок кода, состоящий из нескольких операторов (про блок кода см. выше в самом начале). Примеры:    Оператор switch используется тогда, когда на основе значения переменной необходимо выполнить тот или иной код. Синтаксис оператора switch следующий:    Выражение должно быть строкового (string), целочисленного (byte, int и т.д.) или перечисляемого (enum) типа. Константы и т.д. должны быть того же типа, что и выражение (точнее, выражение должно неявно приводится к типу и т.д.). Секция «default ... break» может отсутствовать. Оператор switch работает следующим образом:  1) вычисляется значение ;  2) среди case блоков выполняется тот, у которого совпало со значением ;  3) если такого блока нет, то выполняется блок по умолчанию «default ... break»;  4) иначе, если блока по умолчанию нет, происходит ошибка времени выполнения.  Пример использования оператора switch: | | |
| 8. | Операторы циклов while, do … while и for и их назначение. Ключевые слова break и continue и их назначение. Примеры. | |
| (все примеры взяты РТ4, если подумать, то можно сделать хорошие примеры) | | |
| Часто встаѐт задача многократного повторения определѐнных действий (например, для обработки данных). Для реализации решений задач такого типа во многих языках предусмотрены так называемые циклы. Рассмотрим синтаксис цикла while:    Здесь – это выражение, которое может принимать значение либо true, либо false; – это оператор (или группа операторов заключѐнных в фигурные скобки), которые будут постоянно выполняться, пока будет истинным (true). Пример:    В результате в текстовом поле будет выведены числа: 1.1, 1.3, 1.5, ..., 5.5. Цикл данного типа может и не выполнится никогда (в данном случае, если изначально будет a > b). Рассмотрим теперь синтаксис цикла do…while:    Здесь – оператор или последовательность операторов, которые постоянно буду выполняться, пока будет истинным (true). Операторы обязательно выполнятся хотя бы раз, т.к. условие проверяется в самом конце. Также обратите внимание: в конце цикла do…while стоит точка с запятой «;». Пример:    В результате в текстовое поле будет выведено: 1.1 . Перейдѐм к рассмотрению синтаксис цикла for:    Цикл for работает следующим образом: 1) Сначала выполняется оператор в секции . Чаще всего здесь стоит объявление новой переменной «переменной цикла» и еѐ инициализация. Допускается опускать этот оператор (секцию), но разделитель «;» всѐ равно ставится. 2) Потом проверяется . Если оно истинно (true), то начинает выполняться основное тело цикла . Иначе начинают выполняться операторы, идущие после цикла (т.е. тело цикла не выполняется). Чаще всего связано с переменной цикла. Секция также может быть пустой. 3) После выполнения тела цикла выполняется секция . В этой секции чаще всего изменяется переменная цикла. Эта секция также может быть пустой. 4) Далее снова всѐ повторяется с пункта (2). Примеры:   Вариант Ответа 2 **Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:**   * **for** * **foreach** * **while** * **do...while**   **Цикл for имеет следующее формальное определение:**   1. **for ([инициализация счетчика]; [условие]; [изменение счетчика])** 2. **{** 3. **// действия** 4. **}**   **Рассмотрим стандартный цикл for:**   1. **for (int i = 0; i < 9; i++)** 2. **{** 3. **Console.WriteLine($"Квадрат числа {i} равен {i\*i}");** 4. **}**   **Первая часть объявления цикла - int i = 0 - создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и другой числовой тип, например, float. И перед выполнением цикла его значение будет равно 0. В данном случае это то же самое, что и объявление переменной.**  **Вторая часть - условие, при котором будет выполняться цикл. Пока условное выражение возвращает true, будет выполняться цикл. В данном случае цикл будет выполняться, пока счетчик i не достигнет 9.**  **И третья часть - приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i--.**  **В итоге блок цикла сработает 9 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.**  Цикл do  **В цикле do сначала выполняется код цикла, а потом происходит проверка условия в инструкции while. И пока это условие истинно, цикл повторяется. Например:**   1. **int i = 6;** 2. **do** 3. **{** 4. **Console.WriteLine(i);** 5. **i--;** 6. **}** 7. **while (i > 0);**   **Здесь код цикла сработает 6 раз, пока i не станет равным нулю. Но важно отметить, что цикл do гарантирует хотя бы единократное выполнение действий, даже если условие в инструкции while не будет истинно. То есть мы можем написать:**   1. **int i = -1;** 2. **do** 3. **{** 4. **Console.WriteLine(i);** 5. **i--;** 6. **}** 7. **while (i > 0);**   **Хотя у нас переменная i меньше 0, цикл все равно один раз выполнится.**  Цикл while  **В отличие от цикла do цикл while сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:**   1. **int i = 6;** 2. **while (i > 0)** 3. **{** 4. **Console.WriteLine(i);** 5. **i--;** 6. **}**   Операторы continue и break  **Иногда возникает ситуация, когда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором break.**  **Например:**   1. **for (int i = 0; i < 9; i++)** 2. **{** 3. **if (i == 5)** 4. **break;** 5. **Console.WriteLine(i);** 6. **}**   **Хотя в условии цикла сказано, что цикл будет выполняться, пока счетчик i не достигнет значения 9, в реальности цикл сработает 5 раз. Так как при достижении счетчиком i значения 5, сработает оператор break, и цикл завершится.**  **Теперь поставим себе другую задачу. А что если мы хотим, чтобы при проверке цикл не завершался, а просто пропускал текущую итерацию. Для этого мы можем воспользоваться оператором continue**   1. **for (int i = 0; i < 9; i++)** 2. **{** 3. **if (i == 5)** 4. **continue;** 5. **Console.WriteLine(i);** 6. **}**   **В этом случае цикл, когда дойдет до числа 5, которое не удовлетворяет условию проверки, просто пропустит это число.** | | |
| 9. | Одномерные и многомерные массивы и работа с ними. Универсальная коллекция List<T>, еѐ создание и работа с ней. Отличие одномерных массивов от List<T>. Примеры. | |
| Массивы используются для хранения данных одного типа. В массивах данные хранятся в оперативной памяти последовательно и доступ к ним осуществляется по их порядковому номеру (индексу) в последовательности. В C#, как и в подавляющем большинстве языков программирования, индексация элементов начинается с нуля.  **Одномерные массивы**  Рассмотрим основной синтаксис создания одномерного массива:    Пример:      Для обращения к элементам массива используют следующий синтаксис:    Пример:    Конструкция с «$» в последней строке называется интерполяцией строки. Это конструкция говорит, что дальше идѐт строка, в которой на местах с фигурными скобками нужно вставить соответствующие значения переменных. Фигурные скобки называют элементами форматирования.  Если нужно скопировать один массив в другой, то для этого нужно воспользоваться стандартным методом CopyTo(), вызов которого выглядит так:    **Многомерные массивы**  Синтаксис создания многомерных массивов аналогичен синтаксису создания одномерных массивов:    В первых квадратных скобках «[, … ,]» количество пустых мест соответствует количеству «размеров», указанных во вторых квадратных скобках. Здесь <размер 1>, …, <размер n> – это размер массива по 1, 2, ... , n измерению соответственно.  Примеры:    **Универсальная коллекция List<T>**  List<T> - своего рода контейнер для хранения данных. Универсальная коллекция List не является по своей сути списком, хотя и реализует некоторые операции, характерные для списка (вставка, удаление и добавление в конец элемента). Это на самом деле массив, который может динамически менять свою длину (что отличает его от обычного одномерного массива). Также, так как List является классом (универсальным классом), то это ссылочный тип и новые объекты данного типа создаются с помощью оператора new.  Синтаксис создания:    Пример: | | |
| 10. | Пользовательские функции и методы. Передача параметров по значению. Передача параметров по ссылке и ключевое слово ref. Тип void и его назначение. Возвращение результата через аргументы и ключевое слово out. Понятие рекурсии. Примеры. | |
| Под **функцией** в программировании понимается обособленный блок кода, способный решать, как правило, одну задачу (класс однотипных задач). Написание функций позволяет решить следующие задачи:  1. разделить программу на понятные логические блоки;  2. не писать многократно один и тот же код;  3. упростить отладку программы, т.к. можно по отдельности проверить работу каждой функции;  4. самодокументировать программный код, давая понятные имена функциям;  5. позволяет создавать богатые библиотеки для решения часто встречаемых на практике задач.  Рассмотрим общий синтаксис объявления функций:    Если функция возвращает значение, то среди операторов обязан присутствовать оператор return (ключевое слово языка). Его синтаксис:    При его выполнении работа функции завершается и вызывающей программе возвращается значение выражения, стоящего после return.  **ref**  Чтобы функция могла менять через аргументы значения в вызывающе программе, используются так называемые ссылки и передача аргументов по ссылке. Для этого в C# используется ключевое слово ref (сокращение от reference «ссылка»). Это ключевое слово ставится:  1. перед типом аргумента в объявлении функции;  2. перед соответствующим аргументом при вызове функции.  Пример:    **void**  Довольно часто функция не должна ничего возвращать (например, функция красивого вывода массива на экран). Для обозначения этого факта в языке C# введѐн специальный тип void (перевод: «пустой», «отсутствие чего-либо»). Этот тип может использоваться только как тип возвращаемого результата.  Пример:    **out**  Для возвращения нескольких значений из функции через аргументы используется другое ключевое слово out (перевод: «наружу»). Синтаксис тот же, что и у ref.  Пример:      **Рекурсия**  Рекурсией в программировании называется вызов функции самой себя. Эта методика применяется тогда, когда решение большой и трудной задачи можно свести к решению более мелких, но похожих задач.  Пример:    В данном случае мы имеем задачу нахождения суммы первых n-чисел, начиная с 1. Так мы сводим все к решению более мелких задач типа “ sum(n) = sum(n-1)+n” – сложению и нахождению sum(n-1). Так можно продолжать пока в итоге мы не придѐм к нахождению sum(1), что можно легко решить и получить 1. | | |
| 11. | Делегаты: их синтаксис и назначение. Универсальные делегаты Action, Predicate и Func: их синтаксис и назначение. Примеры. | |
| Зачастую в программировании при решении определѐнных задач встаѐт потребность в передачи функций в качестве параметра в другие функции. Это позволяет:  1. обеспечить универсальность разрабатываемых алгоритмов;  2. параметризовать алгоритмы дополнительными действиями, меняющими или расширяющими их возможности.  Для решения такого класса задач в языке C# был добавлен специальный пользовательский тип «делегат» (ссылочный тип).  Синтаксис:    Делегат Action используется для передачи функций, которые не должны ничего возвращать. Синтаксис:    Делегат Predicate используется для передачи функций, которые принимают в качестве аргумента один параметр заданного типа и должны возвращать логическое значение. Синтаксис:    Делегат Func используется для передачи функций, которые должны возвращать значение, указанного типа (но не void!). Синтаксис:    Пример: | | |
| 12. | Динамическое создание интерфейса. Программное создание элементов, их настройка и размещение. Примеры. | |
| Этапы:   1. Создание экземпляра элемента интерфейса; 2. Настройка параметров его отображения (размера, положения и т.д.) и иных параметров; 3. Добавление элемента в элемент-контейнер, находящийся на форме, для его дальнейшего отображения на экране.   Рассмотрим все этапы более подробно.  Первый этап:  Для создания элемента интерфейса используется следующая конструкция:    Примеры:    Второй этап:  Для задания размеров элемента используются стандартные для многих элементов свойства Width и Height. Пример:    Для того, чтобы задать положение элемента относительно границ контейнера сначала необходимо установить свойства горизонтального HorizontalAlignment и вертикального VerticalAlignment выравнивания в значения HorizontalAlignment.Left и VerticalAlignment.Top соответственно:    Пример:    Далее необходимо задать положение элемента используется стандартное для многих элементов свойство Margin (отступы). Свойство Margin имеет тип Thickness (толщина). Объект типа Thickness имеет свойства Left, Top, Right, Bottom, которые определяют положение элемента относительно левой, верхней, правой и нижней границы содержащего элемент контейнера. Для задания отступов используется следующая конструкция:    Примеры:      Третий этап:  Основные элементы-контейнеры:   1. Grid – позволяет выравнивать элементы как по сетке, так и произвольно (используется чаще всего); 2. Canvas – позволяет рисовать фигуры, добавленные для отрисовки; 3. StackPanel – выравнивает все элементы интерфейса по ширине контейнера сверху в низ в порядке добавления; 4. WrapPanel – располагает все элементы интерфейса слева на право и далее, при заполнении строки, сверху вниз.   Процесс добавления элементов интерфейса в перечисленные выше контейнеры одинаков:    Пример: | | |
| 13. | Графические возможности WPF. Элемент (контейнер) canvas и работа с ним. Основные графические примитивы, их создание, настройка и размещение. Примеры. | |
| Рисование с помощью библиотеки WPF выглядит так же, как и динамическое создание пользовательского интерфейса и состоит из тех же трѐх этапов. На первом этапе создаются фигуры для отрисовки. Для рисования можно использовать фигуры следующих типов:   1. Ellipse – эллипс 2. Rectangle – прямоугольник 3. Line – прямая линия 4. Polyline – ломаная линия 5. Polygon – многоугольник (ломаная линия, у которой первая и последняя точки соединены, а внутренняя область может быть закрашена).   После создания эллипса нужно указать его ширину и высоту (диаметры):    За заливку внутренней области фигур отвечает свойство Fill (заливка), которое имеет тип Brush (кисть). По умолчанию это свойство имеет значение null (специальное значение, указывающее на отсутствие объекта), что говорит об отсутствии заливки.  В WPF кисти бывают следующих видов (типов):   1. SolidColorBrush – заливка сплошным цветом; 2. ImageBrush – заливка изображением; 3. DrawingBrush – заливка рисунком, составленным из других фигур; 4. LinearGradientBrush – заливка линейным градиентом; 5. RadialGradientBrush – заливка радиальным градиентом; 6. TileBrush – заливка мозаикой на основе изображения.   Рассмотрим создание кисти SolidColorBrush на примере:    Здесь Color (пер. цвет) – это структура данных, у которой есть множество полезных методов для работы с цветом. Например, метод FromRgb позволяет получить цвет на основе смеси красного, зелѐного и синего. На кодирование каждой составляющей цвета отводится 1 байт, т.е. допустимы значения от 0 (отсутствие составляющей) до 255 (максимальное присутствие составляющей). Также есть метод FromArgb, который позволяет помимо цвета задать и его прозрачность.  Более того, существует огромная коллекция заранее заготовленных кистей заданного цвета, собранных в одном классе Brushes.  Для задания цвета границы фигуры используется свойство Stroke (штрих), которое также имеет тип Brush. По умолчанию это свойство имеет значение null, что говорит об отсутствии границы. Для задания толщины контура используется свойство StrokeThickness (double). Толщина задаѐтся в пикселях.  Свойство StrokeDashArray позволяет сделать границу фигуры пунктирной. Это свойство является коллекцией вещественных чисел. Числа с чѐтными индексами определяют длину тире, а с нечѐтными – пробелы между ними. Числа добавляются в коллекцию с помощью стандартного метода Add.  **Canvas**  Canvas - элемент-контейнер, являющийся холстом для отрисовки созданных фигур. Чтобы добавить фигуру в этот контейнер необходимо у его свойства Children вызвать метод Add.  Пример:      Стоит обратить внимание на то, что если фигура выходит за границы холста, то она не обрезается, а рисуется поверх формы (или других элементов интерфейса). Чтобы такого не было необходимо включить обрезку по холсту. Для этого используется свойство ClipToBounds холста (bool). Его нужно установить в значение true:    Структура Point (точка) – это стандартная структура для задания положения точек. У объектов данного типа есть свойства X и Y (координаты точки), которые можно считывать и изменять. | | |
| 14. | Понятие класса. Синтаксис определения класса. Члены класса и модификаторы доступа. Создание и работа с объектами класса. Примеры. | |
| Классы являются пользовательским типом данных. Классы применяются для объединения данных и функций для работы с ними в одну оболочку. Это позволяет добиться следующих эффектов:   1. Классы позволяют сгруппировать данные по их родству. Т.е. если данные описывают один объект, то целесообразно их сгруппировать в виде класса, чтобы и в программном коде с ними можно было работать как единое целое; 2. Классы упрощают чтение программы, так как доступ к данным и к методам, работающим с ними, осуществляется через имя объекта, созданного на основе класса. Не надо больше помнить какие разрозненные данные отвечают за что-то конкретное и как с ними работать. Всѐ в одном флаконе; 3. Правильно написанный класс позволяют создавать объекты, которые сохраняет корректность своего внутреннего состояния на протяжении всей работы программы. Пользователь не может залезть внутрь объекта и сделать там всѐ, что ему хочется, и поломать из-за этого дальнейшую работу объекта класса и программы.   После определения класса можно создавать сколько угодно объектов данного типа, т.е. класс – это тип данных (не существует в оперативной памяти), а объект – это нечто, что создано по «определению класса» и лежит в оперативной памяти.  Синтаксис определения класса:    Модификаторы доступа применяются для контроля доступа к той или иной сущности (т.е. где в коде можно использовать ту или иную сущность, а где нет).  Объекты класса создаются с помощью ключевого слова new:    Пример:    Доступ к полям и методам объекта осуществляется через имя объекта, после которого идѐт точка, а далее имя члена в объявлении класса:    В классе при объявлении переменной можно сразу присвоить ей значение. Так как порядок инициализации переменных заранее не известен, то запрещено при инициализации одной переменной класса использовать другую переменную из того же класса. В других же случаях вы можете пользоваться переменными, константами и типами, которые находятся вне класса, а также константами и типами, которые находятся внутри класса.  Пример: | | |
| 15. ДОДЕЛАТЬ | Понятие перегрузки методов класса. Для чего используется перегрузка методов. Конструкторы и их назначение. Свойства: синтаксис и назначение. Отличие свойств от методов и полей. Примеры.  * Перегрузка - это когда метод уже написан уже существует, но требуется своя реализация со своими входными данными и всё такое * Начальное использование перегрузки - это перегрузка операторов ввода/вывода и арифметических операторов * Конструктор - это метод, который запускается во время создания экземпляра класса * Метод это функция, принадлежащая только классу, в котором она написана и её потомкам (наследование), и она может быть как внутренней (privat) так и public - открытой * Если она приватная, значит её можно вызывать только внутри методов данного класса * А пользователь снаружи не имеет к ней доступа (и не должен иметь) * Поле - почти всегда приватное, и что бы задать или получить значение из него, создаются специальные геттеры (например метод с именем get\_value) и сеттеры (set\_value). Это нужно как раз таки, что бы конечный программист, который будет использовать данную библиотеку или класс не имел доступа к внутренностям кода и не имел возможности использовать этот код неправильно * Свойство же всегда паблик и его конечный программист (его тоже называют пользователем, потому что мы по-сути тоже делаем продукт, которым кто-то должен пользоваться) * Может изменять его свободно и иметь к нему доступ | |
| В классе мы можем вводить свои методы в любом количестве. Например, метод “print” выводит на экран какую | | |
| 16. | Чтение и запись текстовых файлов. Чтение и запись бинарных файлов. Примеры. | |
| **Чтение и запись текстовых файлов**  Для работы с текстовыми файлами используется объект “streamwriter”, который позволяет записывать текстовые файлы.  Синтаксис:    Где “write” - название объекта, “new” существует для создания нового объекта, а в () пишется путь к файлу для записи. Два “\” в пути к файлу пишется для того, чтобы экранировать символ.  Далее с помощью функции “write” в текстовый файл записывается необходимая нам информация, а после файл необходимо закрыть с помощью “close()”. После запуска программы, записанные нами данные должны оказаться в указанном текстовом файле.  Пример:    Итог работы программы:  Для того, чтобы записывать данные с новой строки, вместо “write” используется “writeline”.  Чтение текстовых файлов:  Для считывания данных текстового файла используется “streamreader”. Для этого класса существуют такие методы как “readtoend”, что позволяет считать всю информацию текстового файла, “readline” - для считывания одной строки и “read” - для считывания одного символа.  Пример:  где tb - элемент textbox.  Итог работы приложения:    **Чтение и запись бинарных файлов**  Для записи бинарных файлов используется “BinaryWriter”.  Синтаксис:    Где “bw” - название объекта, “new” существует для создания нового объекта, а в () записывается название файла, в который будет записываться информация. Для бинарных файлов лучше создавать объект “file” (пример ниже).    Для записи используется метод “write”. Ни в коем случае нельзя забывать закрыть файл.  Пример:  Итог работы программы:    мы видим, что записалось 4 байта.  Для чтения бинарных файлов используется “binaryreader”.  Синтаксис:    как мы видим, записывается аналогично “binarywriter”.  Для чтения бинарного файла, необходимо знать тип, который туда записан, потому что для чтения бинарных файлов существуют различные методы, например “readint32”, “readbytes” и т.д..  Пример:    Итог работы приложения:    Консультация 42:45- https://events.webinar.ru/3865279/8829465/record-new/9035269 | | |

**11.** Делегаты Action, Predicate и Func

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются **Action**, **Predicate** и **Func**.

Action

Делегат Action является обобщенным, принимает параметры и возвращает значение void:

1. public delegate void Action<T>(T obj)

Данный делегат имеет ряд перегруженных версий. Каждая версия принимает разное число параметров: от Action<in T1> до Action<in T1, in T2,....in T16>. Таким образом можно передать до 16 значений в метод.

Как правило, этот делегат передается в качестве параметра метода и предусматривает вызов определенных действий в ответ на произошедшие действия. Например:

1. static void Main(string[] args)
2. {
3. Action<int, int> op;
4. op = Add;
5. Operation(10, 6, op);
6. op = Substract;
7. Operation(10, 6, op);
8. Console.Read();
9. }
10. static void Operation(int x1, int x2, Action<int, int> op)
11. {
12. if (x1 > x2)
13. op(x1, x2);
14. }
15. static void Add(int x1, int x2)
16. {
17. Console.WriteLine("Сумма чисел: " + (x1 + x2));
18. }

### Predicate

Делегат Predicate<T>, как правило, используется для сравнения, сопоставления некоторого объекта T определенному условию. В качестве выходного результата возвращается значение true, если условие соблюдено, и false, если не соблюдено:

1. Predicate<int> isPositive = delegate (int x) { return x > 0; };
2. Console.WriteLine(isPositive(20));
3. Console.WriteLine(isPositive(-20));

В данном случае возвращается true или false в зависимости от того, больше нуля число или нет.

### Func

Еще одним распространенным делегатом является **Func**. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

1. TResult Func<out TResult>()
2. TResult Func<in T, out TResult>(T arg)
3. TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2)
4. TResult Func<in T1, in T2, in T3, out TResult>(T1 arg1, T2 arg2, T3 arg3)
5. TResult Func<in T1, in T2, in T3, in T4, out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
6. //...........................................

Данный делегат также часто используется в качестве параметра в методах:

1. static void Main(string[] args)
2. {
3. Func<int, int> retFunc = Factorial;
4. int n1 = GetInt(6, retFunc);
5. Console.WriteLine(n1); // 720
7. int n2 = GetInt(6, x=> x \*x);
8. Console.WriteLine(n2); // 36
9. Console.Read();
10. }
12. static int GetInt(int x1, Func<int, int> retF)
13. {
14. int result = 0;
15. if (x1 > 0)
16. result = retF(x1);
17. return result;
18. }
19. static int Factorial(int x)
20. {
21. int result = 1;
22. for (int i = 1; i <= x; i++)
23. {
24. result \*= i;
25. }
26. return result;
27. }

# **12. Пошаговое руководство. Создание и использование динамических объектов (C# и Visual Basic)**

Динамические объекты предоставляют такие элементы, как свойства и методы, во время выполнения, а не во время компиляции. Это позволяет создавать объекты для работы со структурами, не соответствующими статическому типу или формату. Например, можно использовать динамический объект для ссылки на модель DOM HTML, которая может содержать любую комбинацию допустимых элементов и атрибутов разметки HTML. Поскольку каждый документ HTML является уникальным, элементы для конкретного документа HTML определяются во время выполнения. Наиболее распространенный способ ссылки на атрибут элемента HTML заключается в передаче имени этого атрибута в метод GetProperty элемента. Для ссылки на атрибут id элемента HTML <div id="Div1"> следует сначала получить ссылку на элемент <div>, а затем использовать divElement.GetProperty("id"). При использовании динамического объекта можно сослаться на атрибут id в виде divElement.id.

Динамические объекты обеспечивают удобный доступ к динамическим языкам, таким как IronPython и IronRuby. С помощью динамического объекта можно ссылаться на динамический скрипт, интерпретируемый во время выполнения.

Ссылка на динамический объект выполняется с помощью позднего связывания. В C# тип объектов с поздним связыванием указывается как dynamic. В Visual Basic тип объектов с поздним связыванием указывается как Object. Дополнительные сведения см. в разделах [dynamic](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/reference-types) и [Раннее и позднее связывание](https://docs.microsoft.com/ru-ru/dotnet/visual-basic/programming-guide/language-features/early-late-binding/).

Вы можете создавать настраиваемые динамические объекты, используя классы из пространства имен [System.Dynamic](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic). Например, можно создать объект [ExpandoObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.expandoobject) и задать члены этого объекта во время выполнения. Также можно создать собственный тип, наследующий класс [DynamicObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject). Затем для обеспечения динамических функциональных возможностей во время выполнения можно переопределить члены класса [DynamicObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject).

## **Создание пользовательского динамического объекта**

В первом пошаговом руководстве определяется пользовательский динамический объект, выполняющий поиск по содержимому текстового файла. Динамическое свойство указывает искомый текст. Например, если в вызывающем коде указано dynamicFile.Sample, динамический класс возвращает общий список строк, содержащий все строки из файла, которые начинаются со слова "Sample". При поиске не учитывается регистр. Динамический класс также поддерживает два дополнительных аргумента. Первый аргумент — это значение перечисления параметра поиска, задающее, где динамический класс должен искать соответствия: в начале строки, в конце строки или в любом месте строки. Второй аргумент задает, что динамический класс должен перед поиском отсекать начальные и конечные пробелы в каждой строке. Например, если в вызывающем коде указано dynamicFile.Sample(StringSearchOption.Contains), динамический класс выполняет поиск слова "Sample" в любом месте строки. Если в вызывающем коде указано dynamicFile.Sample(StringSearchOption.StartsWith, false), динамический класс выполняет поиск слова "Sample" в начале каждой строки и не удаляет начальные и конечные пробелы в строках. По умолчанию динамический класс выполняет поиск соответствия в начале каждой строки, предварительно удаляя начальные и конечные пробелы.

### **Создание пользовательского динамического класса**

1. Запустите Visual Studio.
2. Выберите **Создать новый проект**.
3. В диалоговом окне **Создание нового проекта** выберите C# или Visual Basic, выберите **Консольное приложение**, а затем нажмите кнопку **Далее**.
4. В диалоговом окне **Настройка нового проекта** введите значение DynamicSample для параметра **Имя проекта** и нажмите кнопку **Далее**.
5. В диалоговом окне **Дополнительные сведения** выберите значение **.NET 5.0 (текущая)** для параметра **Целевая платформа**, а затем нажмите кнопку **Создать**.Создается новый проект.
6. В **обозревателе решений** щелкните проект DynamicSample правой кнопкой мыши и выберите **Добавить** > **Класс**. В поле **Имя** введите ReadOnlyFile, а затем нажмите кнопку **Добавить**.Будет добавлен новый файл, содержащий класс ReadOnlyFile.
7. В верхней части файла *ReadOnlyFile.cs* или *ReadOnlyFile.vb* добавьте следующий код для импорта пространств имен [System.IO](https://docs.microsoft.com/ru-ru/dotnet/api/system.io) и [System.Dynamic](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic).

using System.IO;

using System.Dynamic;

1. Пользовательский динамический объект использует перечисление для определения условия поиска. Перед оператором класса добавьте следующее определение перечисления.

public enum StringSearchOption

{

StartsWith, Contains, EndsWith

}

1. Обновите оператор класса, чтобы он наследовал класс DynamicObject, как показано в следующем примере кода.

public enum StringSearchOption { StartsWith, Contains, EndsWith }

1. Добавьте в класс ReadOnlyFile следующий код, чтобы задать закрытое поле для пути к файлу и конструктор для класса ReadOnlyFile.

// Store the path to the file and the initial line count value.

private string p\_filePath;

// Public constructor. Verify that file exists and store the path in

// the private variable.

public ReadOnlyFile(string filePath)

{

if (!File.Exists(filePath))

{

throw new Exception("File path does not exist.");

}

p\_filePath = filePath;

}

1. Добавьте следующий метод GetPropertyValue в класс ReadOnlyFile. Метод GetPropertyValue принимает в качестве входных данных условие поиска и возвращает строки текстового файла, соответствующие этому условию. Динамический метод, предоставленный классом ReadOnlyFile, вызывает метод GetPropertyValue для извлечения соответствующих результатов.

public List<string> GetPropertyValue(string propertyName,

StringSearchOption StringSearchOption = StringSearchOption.StartsWith,

bool trimSpaces = true)

{

StreamReader sr = null;

List<string> results = new List<string>();

string line = "";

string testLine = "";

try

{

sr = new StreamReader(p\_filePath);

while (!sr.EndOfStream)

{

line = sr.ReadLine();

// Perform a case-insensitive search by using the specified search options.

testLine = line.ToUpper();

if (trimSpaces) { testLine = testLine.Trim(); }

switch (StringSearchOption)

{

case StringSearchOption.StartsWith:

if (testLine.StartsWith(propertyName.ToUpper())) { results.Add(line); }

break;

case StringSearchOption.Contains:

if (testLine.Contains(propertyName.ToUpper())) { results.Add(line); }

break;

case StringSearchOption.EndsWith:

if (testLine.EndsWith(propertyName.ToUpper())) { results.Add(line); }

break;

}

}

}

catch

{

// Trap any exception that occurs in reading the file and return null.

results = null;

}

finally

{

if (sr != null) {sr.Close();}

}

return results;

}

1. После метода GetPropertyValue добавьте следующий код, чтобы переопределить метод [TryGetMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.trygetmember) класса [DynamicObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject). Метод [TryGetMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.trygetmember) вызывается при запросе члена динамического класса без указания аргументов. Аргумент binder содержит сведения об элементе, на который дается ссылка, а аргумент result ссылается на результат, возвращенный для указанного элемента. Метод [TryGetMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.trygetmember) возвращает логическое значение true, если запрошенный элемент существует. В противном случае возвращается false.

// Implement the TryGetMember method of the DynamicObject class for dynamic member calls.

public override bool TryGetMember(GetMemberBinder binder,

out object result)

{

result = GetPropertyValue(binder.Name);

return result == null ? false : true;

}

1. После метода TryGetMember добавьте следующий код, чтобы переопределить метод [TryInvokeMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.tryinvokemember) класса [DynamicObject](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject). Метод [TryInvokeMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.tryinvokemember) вызывается при запросе члена динамического класса с аргументами. Аргумент binder содержит сведения об элементе, на который дается ссылка, а аргумент result ссылается на результат, возвращенный для указанного элемента. Аргумент args содержит массив аргументов, передаваемых в элемент. Метод [TryInvokeMember](https://docs.microsoft.com/ru-ru/dotnet/api/system.dynamic.dynamicobject.tryinvokemember) возвращает логическое значение true, если запрошенный элемент существует. В противном случае возвращается false. Пользовательская версия метода TryInvokeMember ожидает, что первый аргумент будет значением из перечисления StringSearchOption, заданного на предыдущем шаге. Метод TryInvokeMember ожидает, что второй аргумент будет логическим значением. Если один или оба элемента имеют допустимые значения, они передаются в метод GetPropertyValue для получения результатов.

// Implement the TryInvokeMember method of the DynamicObject class for

// dynamic member calls that have arguments.

public override bool TryInvokeMember(InvokeMemberBinder binder,

object[] args,

out object result)

{

StringSearchOption StringSearchOption = StringSearchOption.StartsWith;

bool trimSpaces = true;

try

{

if (args.Length > 0) { StringSearchOption = (StringSearchOption)args[0]; }

}

catch

{

throw new ArgumentException("StringSearchOption argument must be a StringSearchOption enum value.");

}

try

{

if (args.Length > 1) { trimSpaces = (bool)args[1]; }

}

catch

{

throw new ArgumentException("trimSpaces argument must be a Boolean value.");

}

result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

return result == null ? false : true;

}

1. Сохраните и закройте файл.

### **Создание примера приложения, в котором применяется пользовательский динамический объект**

1. В **обозревателе решений** дважды щелкните файл *Program.vb*, если используется Visual Basic, или файл *Program.cs*, если используется Visual C#.
2. Добавьте следующий код в процедуру Main, чтобы создать экземпляр класса ReadOnlyFile для файла *TextFile1.txt*. В этом коде используется позднее связывание для вызова динамических элементов и извлечения строк текста, которые содержат строку "Customer".

dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");

foreach (string line in rFile.Customer)

{

Console.WriteLine(line);

}

Console.WriteLine("----------------------------");

foreach (string line in rFile.Customer(StringSearchOption.Contains, true))

{

Console.WriteLine(line);

}

1. Сохраните файл и нажмите клавиши CTRL+F5 для сборки и запуска приложения.

# **13.** Компоновка.

## Введение в компоновку

Чтобы перейти уже непосредственно к созданию красивых интерфейсов и их компонентов, сначала необходимо познакомиться с компоновкой. Компоновка (layout) представляет собой процесс размещения элементов внутри контейнера. Возможно, вы обращали внимание, что одни программы и веб-сайты на разных экранах с разным разрешением выглядят по-разному: где-то лучше, где-то хуже. В большинстве своем такие программы используют жестко закодированные в коде размеры элементов управления. WPF уходит от такого подхода в пользу так называемого "резинового дизайна", где весь процесс позиционирования элементов осуществляется с помощью компоновки.

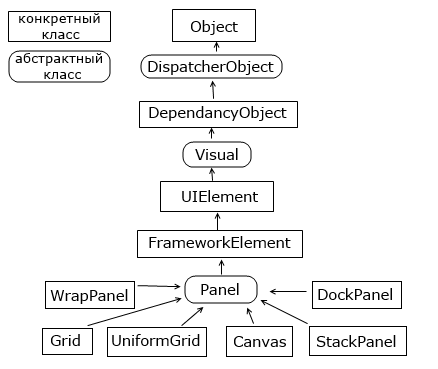
Благодаря компоновке мы можем удобным нам образом настроить элементы интерфейса, позиционировать их определенным образом. Например, элементы компоновки в WPF позволяют при ресайзе - сжатии или растяжении масштабировать элементы, что очень удобно, а визуально не создает всяких шероховатостей типа незаполненных пустот на форме.

В WPF компоновка осуществляется при помощи специальных контейнеров. Фреймворк предоставляет нам следующие контейнеры: **Grid**, **UniformGrid**, **StackPanel**, **WrapPanel**, **DockPanel** и **Canvas**.

Различные контейнеры могут содержать внутри себя другие контейнеры. Кроме данных контейнеров существует еще ряд элементов, такие как TabPanel, которые могут включать другие элементы и даже контейнеры компоновки, однако на саму компоновку не столь влияют в отличие от выше перечисленных. Кроме того, если нам не хватает стандартных контейнеров, мы можем определить свои с нужной нам функциональностью.

Контейнеры компоновки позволяют эффективно распределить доступное пространство между элементами, найти для него наиболее предпочтительные размеры.

Все выше перечисленные контейнеры компоновки наследуются от абстрактного класса Panel, а само дерево наследования можно представить следующим образом:



Наглядным примером применения компоновки может служить сделанное в первой главе приложение калькулятора, где мы не указываем явным образом размеры кнопок, а полагаемся на контейнер Grid, который и осуществляет всю расстановку и масштабирование дочерних элементов.

В WPF при компоновке и расположении элементов внутри окна нам надо придерживаться следующих принципов:

* Нежелательно указывать явные размеры элементов (за исключеним минимальных и максимальных размеров). Размеры должны определяться контейнерами.
* Нежелательно указывать явные позицию и координаты элементов внутри окна. Позиционирование элементов всецело должно быть прерогативой контейнеров. И контейнер сам должен определять, как элемент будет располагаться. Если нам надо создать сложную систему компоновки, то мы можем вкладывать один контейнер в другой, чтобы добиться максимально удобного расположения элементов управления.

### Процесс компоновки

Процесс компоновки проходит два этапа: измерение (measure) и расстановка (arrange). На этапе измерения контейнер производит измерение предпочтительного для дочерних элементов места. Однако не всегда контейнер имеет достаточно места, чтобы расставить все элементы по их предпочтительным размером, поэтому их размеры приходится усекать. Затем происходит этап непосредственной расстановки дочерних элементов внутри контейнера.

Теперь рассмотрим контейнеры компоновки подробнее.

# **14.**Классы. Объектно-ориентированное программирование.Классы и объекты

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. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке, у которого есть имя, возраст, какие-то другие характеристики. То есть некоторый шаблон - этот шаблон можно назвать классом. Конкретное воплощение этого шаблона может отличаться, например, одни люди имеют одно имя, другие - другое имя. И реально существующий человек (фактически экземпляр данного класса) будет представлять объект этого класса.

По умолчанию проект консольного приложения уже 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**:

1. class Person
2. {
3. }

Где определяется класс? Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс Program. То есть файл Program.cs будет выглядеть следующим образом:

1. using System;
2. namespace HelloApp
3. {
4. class Person
5. {
6. }
7. class Program
8. {
9. static void Main(string[] args)
10. {
12. }
13. }
14. }

Вся функциональность класса представлена его членами - полями (полями называются переменные класса), свойствами, методами, событиями. Например, определим в классе Person поля и метод:

1. using System;
2. namespace HelloApp
3. {
4. class Person
5. {
6. public string name; // имя
7. public int age = 18; // возраст
9. public void GetInfo()
10. {
11. Console.WriteLine($"Имя: {name} Возраст: {age}");
12. }
13. }
14. class Program
15. {
16. static void Main(string[] args)
17. {
18. Person tom;
19. }
20. }
21. }

В данном случае класс Person представляет человека. Поле name хранит имя, а поле age - возраст человека. А метод GetInfo выводит все данные на консоль. Чтобы все данные были доступны вне класса Person переменные и метод определены с модификатором public. Поскольку поля фактически те же переменные, им можно присвоить начальные значения, как в случае выше, поле age инициализировано значением 18.

Так как класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе Main определена переменная tom, которая представляет класс Person. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение **null**. Поэтому вначале необходимо создать объект класса Person.

### 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.

#### Конструктор по умолчанию

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.

Выше класс Person не имеет никаких конструкторов. Поэтому для него автоматически создается конструктор по умолчанию. И мы можем использовать этот конструктор. В частности, создадим один объект класса Person:

1. class Person
2. {
3. public string name; // имя
4. public int age; // возраст
6. public void GetInfo()
7. {
8. Console.WriteLine($"Имя: {name} Возраст: {age}");
9. }
10. }
11. class Program
12. {
13. static void Main(string[] args)
14. {
15. Person tom = new Person();
16. tom.GetInfo(); // Имя: Возраст: 0
18. tom.name = "Tom";
19. tom.age = 34;
20. tom.GetInfo(); // Имя: Tom Возраст: 34
21. Console.ReadKey();
22. }
23. }

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";.

**Имя:Возраст: 0**

**Имя: TomВозраст: 34**

#### Создание конструкторов

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

1. class Person
2. {
3. public string name;
4. public int age;
6. public Person() { name = "Неизвестно"; age = 18; } // 1 конструктор
8. public Person(string n) { name = n; age = 18; } // 2 конструктор
10. public Person(string n, int a) { name = n; age = a; } // 3 конструктор
12. public void GetInfo()
13. {
14. Console.WriteLine($"Имя: {name} Возраст: {age}");
15. }
16. }

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

1. static void Main(string[] args)
2. {
3. Person tom = new Person(); // вызов 1-ого конструктора без параметров
4. Person bob = new Person("Bob"); //вызов 2-ого конструктора с одним параметром
5. Person sam = new Person("Sam", 25); // вызов 3-его конструктора с двумя параметрами
6. bob.GetInfo(); // Имя: Bob Возраст: 18
7. tom.GetInfo(); // Имя: Неизвестно Возраст: 18
8. sam.GetInfo(); // Имя: Sam Возраст: 25
9. }

Консольный вывод данной программы:

**Имя: Неизвестно Возраст: 18**

**Имя: Bob Возраст: 18**

**Имя: Sam Возраст: 25**

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

Стоит отметить, что начиная с версии C# 9.0 мы можем сократить вызов конструктора, убрав из него название типа:

1. Person tom = new (); // аналогично new Person();
2. Person bob = new ("Bob"); // аналогично new Person("Bob");
3. Person sam = new ("Sam", 25); // аналогично new Person("Sam", 25);

### 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:

1. class Person
2. {
3. public string name;
4. public int age;
5. public Person() : this("Неизвестно")
6. {
7. }
8. public Person(string name) : this(name, 18)
9. {
10. }
11. public Person(string name, int age)
12. {
13. this.name = name;
14. this.age = age;
15. }
16. public void GetInfo()
17. {
18. Console.WriteLine($"Имя: {name} Возраст: {age}");
19. }
20. }

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:

1. public Person(string name) : this(name, 18)
2. {
3. }

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:

1. Person tom = new Person { name = "Tom", age=31 };
2. 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.

## **15.** Перегрузка методов

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Такая возможность еще называется **перегрузкой методов** (method overloading).

И в языке C# мы можем создавать в классе несколько методов с одним и тем же именем, но разной сигнатурой. Что такое сигнатура? **Сигнатура** складывается из следующих аспектов:

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

Но названия параметров в сигнатуру НЕ входят. Например, возьмем следующий метод:

1. public int Sum(int x, int y)
2. {
3. return x + y;
4. }

У данного метода сигнатура будет выглядеть так: Sum(int, int)

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

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

Например, пусть у нас есть следующий класс:

1. class Calculator
2. {
3. public void Add(int a, int b)
4. {
5. int result = a + b;
6. Console.WriteLine($"Result is {result}");
7. }
8. public void Add(int a, int b, int c)
9. {
10. int result = a + b + c;
11. Console.WriteLine($"Result is {result}");
12. }
13. public int Add(int a, int b, int c, int d)
14. {
15. int result = a + b + c + d;
16. Console.WriteLine($"Result is {result}");
17. return result;
18. }
19. public void Add(double a, double b)
20. {
21. double result = a + b;
22. Console.WriteLine($"Result is {result}");
23. }
24. }

Здесь представлены четыре разных версии метода Add, то есть определены четыре перегрузки данного метода.

Первые три версии метода отличаются по количеству параметров. Четвертая версия совпадает с первой по количеству параметров, но отличается по их типу. При этом достаточно, чтобы хотя бы один параметр отличался по типу. Поэтому это тоже допустимая перегрузка метода Add.

То есть мы можем представить сигнатуры данных методов следующим образом:

Add(int, int)

Add(int, int, int)

Add(int, int, int, int)

Add(double, double)

После определения перегруженных версий мы можем использовать их в программе:

1. class Program
2. {
3. static void Main(string[] args)
4. {
5. Calculator calc = new Calculator();
6. calc.Add(1, 2); // 3
7. calc.Add(1, 2, 3); // 6
8. calc.Add(1, 2, 3, 4); // 10
9. calc.Add(1.4, 2.5); // 3.9
10. Console.ReadKey();
11. }
12. }

Консольный вывод:

**Result is 3**

**Result is 6**

**Result is 10**

**Result is 3.9**

Также перегружаемые методы могут отличаться по используемым модификаторам. Например:

1. void Increment(ref int val)
2. {
3. val++;
4. Console.WriteLine(val);
5. }
6. void Increment(int val)
7. {
8. val++;
9. Console.WriteLine(val);
10. }

В данном случае обе версии метода Increment имеют одинаковый набор параметров одинакового типа, однако в первом случае параметр имеет модификатор ref. Поэтому обе версии метода будут корректными перегрузками метода Increment.

А отличие методов по возвращаемому типу или по имени параметров не является основанием для перегрузки. Например, возьмем следующий набор методов:

1. int Sum(int x, int y)
2. {
3. return x + y;
4. }
5. int Sum(int number1, int number2)
6. {
7. return number1 + number2;
8. }
9. void Sum(int x, int y)
10. {
11. Console.WriteLine(x + y);
12. }

Сигнатура у всех этих методов будет совпадать:

Sum(int, int)

Поэтому данный набор методов не представляет корректные перегрузки метода Sum и работать не будет.

## **16.Чтение и запись бинарных файлов**

Для работы с бинарными файлами предназначена пара классов **BinaryWriter** и **BinaryReader**. Эти классы позволяют читать и записывать данные в двоичном формате.

### Основные метода класса BinaryWriter

* **Close()**: закрывает поток и освобождает ресурсы
* **Flush()**: очищает буфер, дописывая из него оставшиеся данные в файл
* **Seek()**: устанавливает позицию в потоке
* **Write()**: записывает данные в поток

### Основные метода класса BinaryReader

* **Close()**: закрывает поток и освобождает ресурсы
* **ReadBoolean()**: считывает значение bool и перемещает указатель на один байт
* **ReadByte()**: считывает один байт и перемещает указатель на один байт
* **ReadChar()**: считывает значение char, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
* **ReadDecimal()**: считывает значение decimal и перемещает указатель на 16 байт
* **ReadDouble()**: считывает значение double и перемещает указатель на 8 байт
* **ReadInt16()**: считывает значение short и перемещает указатель на 2 байта
* **ReadInt32()**: считывает значение int и перемещает указатель на 4 байта
* **ReadInt64()**: считывает значение long и перемещает указатель на 8 байт
* **ReadSingle()**: считывает значение float и перемещает указатель на 4 байта
* **ReadString()**: считывает значение string. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа int занимает 4 байта, поэтому BinaryReader считает 4 байта и переместит указать на эти 4 байта.

Посмотрим на реальной задаче применение этих классов. Попробуем с их помощью записывать и считывать из файла массив структур:

1. struct State
2. {
3. public string name;
4. public string capital;
5. public int area;
6. public double people;
8. public State(string n, string c, int a, double p)
9. {
10. name = n;
11. capital = c;
12. people = p;
13. area = a;
14. }
15. }
16. class Program
17. {
18. static void Main(string[] args)
19. {
20. State[] states = new State[2];
21. states[0] = new State("Германия", "Берлин", 357168, 80.8);
22. states[1] = new State("Франция", "Париж", 640679, 64.7);
24. string path= @"C:\SomeDir\states.dat";
26. try
27. {
28. // создаем объект BinaryWriter
29. using (BinaryWriter writer = new BinaryWriter(File.Open(path, FileMode.OpenOrCreate)))
30. {
31. // записываем в файл значение каждого поля структуры
32. foreach (State s in states)
33. {
34. writer.Write(s.name);
35. writer.Write(s.capital);
36. writer.Write(s.area);
37. writer.Write(s.people);
38. }
39. }
40. // создаем объект BinaryReader
41. using (BinaryReader reader = new BinaryReader(File.Open(path, FileMode.Open)))
42. {
43. // пока не достигнут конец файла
44. // считываем каждое значение из файла
45. while (reader.PeekChar() > -1)
46. {
47. string name = reader.ReadString();
48. string capital = reader.ReadString();
49. int area = reader.ReadInt32();
50. double population = reader.ReadDouble();
51. Console.WriteLine("Страна: {0} столица: {1} площадь {2} кв. км численность населения: {3} млн. чел.",
52. name, capital, area, population);
53. }
54. }
55. }
56. catch (Exception e)
57. {
58. Console.WriteLine(e.Message);
59. }
60. Console.ReadLine();
61. }
62. }

Итак, у нас есть структура State с некоторым набором полей. В основной программе создаем массив структур и записываем с помощью BinaryWriter. Этот класс в качестве параметра в конструкторе принимает объект Stream, который создается вызовом File.Open(path, FileMode.OpenOrCreate).

Затем в цикле пробегаемся по массиву структур и записываем каждое поле структуры в поток. В том порядке, в каком эти значения полей записываются, в том порядке они и будут размещаться в файле.

Затем считываем из записанного файла. Конструктор класса BinaryReader также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима FileMode.Open: new BinaryReader(File.Open(path, FileMode.Open))

В цикле while считываем данные. Чтобы узнать окончание потока, вызываем метод PeekChar(). Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает -1, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения поле структур в том же порядке, в каком они записывались.

Таким образом, классы BinaryWriter и BinaryReader очень удобны для работы с бинарными файлами, особенно когда нам известна структура этих файлов. В то же время для хранения и считывания более комплексных объектов, например, объектов классов, лучше подходит другое решение - сериализация.