# Вступление  
 you you you you \*Друзья\*, всех приветствую! Поздравляю вас с выходом на финишную прямую нашего курса. Сегодня мы рассматриваем заключительные тематики, а также подводим итоги проведенных занятий. План на сегодня у нас следующий.  
  
# Рассмотреть | Рекурсия |  
Дадим определение \*рекурсии\*, \*рассмотрим\*, что она из себя представляет, а также приведем примеры из жизни.  
  
# Рассмотреть | Рекурсия | Отладка |  
Далее мы поговорим о рекурсии в контексте программирования, рассмотрим ее составные части и решим блок задач. Отдельным пунктом обсудим, что такое \*\*отладка\* программ\*. И примерим два способа отладки на наших программах.  
  
# Рассмотреть | Рекурсия |  
В заключение рассмотрим три полезных принципа разработки программ, а также приведем небольшое собеседование по тематике \*рекурсия\*.  
  
# Рекурсия | Алгоритм |  
Итак, давайте начнем. Для новичков в программировании тематика рекурсии, рекурсивные \*алгоритмы\* обычно дается сложно.  
  
# Рассмотреть | Рекурсия |  
Поэтому перед тем, как говорить о рекурсии в контексте языка программирования, давайте рассмотрим это понятие в более широком смысле и приведем реальные примеры из жизни. Под рекурсией в широком смысле мы понимаем описание объекта или процесса, который находится в этом же самом объекте или же процессе. То есть объект является частью самого себя. Наиболее широкое применение рекурсия имеет в математике и информатике. Но мы с вами начнем знакомство с рекурсией на более простых примерах, чтобы уловить ее основные идеи.  
  
# Рассмотреть | Рекурсия |  
Первый пример, который мы рассмотрим, это техника изображения рекурсии. Суть ее в том, что внутри изображения размещается его уменьшенная копия, как вы можете видеть на картинке справа.  
  
# Абзац  
Внутри этой уменьшенной копии копия предыдущего изображения и так далее теоретически до бесконечности. На практике же до пределов допустимого качества изображения. То есть пока его можно разглядеть на картинке.  
  
# Рекурсия | Друг |  
Второй пример рекурсии, с которым, я думаю, каждый из вас сталкивался, это набор матрешек. Обратите внимание, что здесь выполняется определение рекурсии. Описание объекта, например, одна из матрешек, который находится в этом же объекте, в некоторой более крупной матрешке. Отмечу, что эта рекурсия не бесконечна, поскольку какую бы большую матрешку мы изначально не взяли, допустим, вот которая здесь в правой части экрана, мы дойдем до таких размеров, меньше которых изготовить фигууры уже физически будет невозможно. Третий пример похож на первый. Здесь также применяется техника размещения уменьшенной копии изображения в нем же самом. В этом примере рекурсия также будет ограничена возможностью изобразить мелкие детали рисунка. Подобное изображение можно увидеть при работе за компьютером. Здесь изображено окно программы для удаленного управления с включенной демонстрацией рабочего стола. Если расшарить экран на обоих компьютерах, на управляющем и на управляемом, то можно увидеть подобную картинку. В данном примере, друзья, изображение рабочего стола будет уменьшаться, пока не достигнет размеров буквально в 1 пиксель.  
  
# Рассмотреть |  
И в заключение этой рубрики давайте рассмотрим пример, который показывает принцип самовоспроизведения. Здесь акцент делается не на использовании уменьшенной копии изображения, как мы делали в предыдущих примерах. Здесь же акцент делается на самовоспроизведении, а также одновременном усложнении системы. Базовой фигурой в этом рисунке является пара соединенных линий, которые образуют одну ветку. Ветки большего размера строятся из базовой фигуры. Простая ветка самовоспроизводится и одновременно фигура усложняется, то есть веток становится больше. То есть вторая фигура построена, как вы можете видеть, из трех базовых веток. Первая ветка, вторая и третья в самом низу. Третья фигура уже состоит из семи веток и четвертая заключает в себе 15 веток. По мере добавления в рисунок новых веток они уменьшаются в размерах, чтобы соблюдать пропорции дерева.  
  
# Абзац  
Обратите внимание, что на второй фигуре можно видеть два уровня веток, на третьей – три и на четвертой – четыре уровня веток. Вопрос для любителей математики. А как вы думаете, есть ли какая-то закономерность между количеством уровней и числом веток в дереве? Свои предположения можете написать в чат.  
  
# Друг |  
Итак, друзья, после экскурса в рекурсию в широком смысле, давайте поговорим о ней в контексте программирования.  
  
# Рассмотреть | Функция |  
Дадим более точное определение, рассмотрим, с каких частей она состоит и, самое главное, попытаемся понять, а как же она работает. Для того, чтобы определить рекурсию в программировании, мы задействуем знакомый нам термин \*функции\*.  
  
# Абзац  
Итак, под рекурсией мы будем понимать некоторый процесс, в котором функция вызывает саму себя. Такую функцию мы будем называть рекурсивной, а подобный вызов называется рекурсивным вызовом.  
  
# Рассмотреть | Матрёшка | Друг | Работа | Функция | Openmatryoshka | Код |  
Чтобы было понимание, как это выглядит на практике, давайте рассмотрим фрагмент \*кода\*, который имитирует открытие \*матрешки\*. На экране представлена функция, которая уведомляет, что открывает матрешку и выводит на экран ее размер. Затем можно видеть вызов этой же функции \*openMatryoshka\* и как раз таки этот вызов, друзья, и называется рекурсивным. А сама функция openMatрешка называется рекурсивной функцией. Процесс \*работы\* такой функции мы называем рекурсией. Чтобы все это дело запустить, нам необходимо один раз вызвать функцию open матрешка и передать ей некоторое значение. Параметр функции говорит о суммарном количестве матрешек.  
  
# Ошибка | Рекурсия | Друг |  
Друзья, с точки зрения логики этот пример содержит \*ошибки\*, которые мы исправим далее. Цель этого примера показать основные компоненты рекурсии, которые мы ввели ранее.  
  
# Параметр | Рассмотреть | Рекурсивный | Матрёшка | Друг | Функция | Вызов |  
Следующим этапом давайте рассмотрим их более детально. Первый элемент это \*рекурсивный\* вызов. Давайте отметим себе, что целью рекурсивного \*вызова\* является решение более простой задачи или же более меньшей, чем текущая. Здесь я попытался изобразить набор \*вызовов\*, где на каждом этапе решается более простая задача, чем текущая. Под более простой задачей понимается задача с меньшим объемом вычислений. В данной схеме последним вызовом будет выполняться задача, которая является тривиальной. И упрощать ее уже некуда. Отметьте себе, что такая задача называется базовым случаем. Если говорить про пример с матрешками, то \*базовый случай\* это самая вложенная или самая маленькая матрешка. Вы можете задаться вопросом, а как сделать задачу более простой? Как ее уменьшить? Ведь функция, в которой производится вычисление, никаким образом не меняется. Набор инструкций, из которых состоит тело функции, остается неизменным. Друзья, здесь все дело в параметрах функции. Именно за счет них мы можем влиять на ее поведение. В самом простом случае мы можем уменьшить величину параметров. Если это размер матрешки, как в приведенном примере, то можно просто уменьшить его на 1. Тем самым мы упростим решаемую задачу и приблизим ее к базовому случаю. Вот эта вот функция function может открывать матрешку, может выводить на экран какие-то данные, а может произвести вычисления. Для нас главное, что за счет изменения \*параметра\* мы делаем задачу каждый раз проще. Приводим ее к базовому случаю. Вернемся к примеру. Основное действие функции это имитация открытия матрешки с выводом ее размера.  
  
# Параметр | Рекурсия | Рекурсивный | Матрёшка | Видеть |  
Далее мы \*видим\* рекурсивный вызов – один из основных компонентов рекурсии. Но обратите внимание, что вызов осуществляется с тем же значением параметра size. То есть он никак не меняется. Здесь не выполняется требование, что рекурсивный вызов должен приводить к решению более простой задачи. Давайте подумаем, как можно его оформить, чтобы он приблизил нас к базовому случаю. Чтобы этого достичь, можно сделать рекурсивный вызов со значением параметра, допустим, на единицу меньше. Тем самым мы будем приближаться к базовому случаю можно считать это открытие матрешки размером 1 открытие матрешки единичного размера будем считать тривиальной задачи после выполнения которой нужно остановиться запустим рекурсию для этого вызовем функцию с некоторым значением допустустим равным 5. На экране мы видим процесс открытия матрешек размером от 5 до 1. Однако после открытия матрешки с размером 1, как можно видеть, процедура продолжается. Размеры матрешек уже стали отрицательными. Возникает вопрос, остановится ли когда-нибудь этот процесс. Ответ – нет, не остановится, поскольку он работает как бесконечный цикл. Каких-либо условий для остановки рекурсии в коде мы с вами не писали.  
  
# Параметр | Рекурсия | Рекурсивный | Видеть | Матрёшка | Работа | Функция | Друг | Вызов |  
Друзья, мы подошли ко второй составляющей рекурсии. Это условия для ее остановки. Если этого не делать, то \*рекурсивные вызовы\* будут выполняться бесконечно, что в итоге приведет к зацикливанию или же ошибке во время выполнения. Рекурсию останавливают, когда на очередном вызове становится понятно, что задача стала простейшей. То есть мы подошли к базовому случаю. Обычно на это указывает значение параметра. В этом случае рекурсийный вызов не делается и функцию завершают инструкции return. При необходимости возвращают значение. Вернемся к нашему примеру и посмотрим, а как можно остановить рекурсию в нужный момент. Для этого нужно понять, а каким должно быть значение параметра size, чтобы считать эту задачу тривиальной, или же простейшей, или другими словами базовым случаем. Как мы сказали ранее, эта функция открывает матрешки до единичного размера. Следовательно, открытие матрешки размером 1 мы можем считать базовым случаем. Добавим условие, которое проверит равенство параметра единиц. Если это так, то завершим выполнение функции без рекурсивного вызова. Воспользуемся инструкцией return. Также для информативности добавим соответствующее сообщение о том, что мы достигли самой маленькой матрешки. Посмотрим, каким будет результат работы программы. В окне вывода мы видим последовательность открытия матрешек с размерами от 5 и до единичного размера. На этом рекурсивные вызовы останавливаются и мы видим соответствующее уведомление. Друзья, мы с вами рассмотрели достаточно простой вариант применения рекурсии. Когда функция сначала делает некоторую работу и в конце вызывает себя рекурсивным. Однако, если функция будет вначале иметь рекурсивный вызов, а затем выполнение работы, то ее поведение будет несколько отличаться. Вернемся к нашему примеру. В нем рекурсивный вызов происходит после выполнения основных действий, после вывода на экран информации об открытии очередной матрешки. Давайте немного перепишем функцию так, чтобы сначала шел рекурсивный вызов, а затем выполнялась основная работа. Теперь вывод сообщения на экран находится в конце функции. Вначале мы видим рекурсивный вызов, также с проверкой условий на базовый случай. Посмотрим на результат работы этой функции и сравним его с предыдущим выводом. В консоли мы видим все тот же набор сообщений об открытии матрешек, но в этом случае размеры идут по возрастанию от двойки и до суммарного количества матрешек. Функция у нас вызывается все с тем же аргументом равным 5, однако размеры идут в обратном порядке. Попробуем разобраться, как работает рекурсия в этом случае. Изобразим схематически, что будет происходить при вызове функции с аргументом равным 4. На схеме изображены 4 вызова функции open матрешка. Первый вызов – это первоначальный вызов функции с аргументом равным четырем.  
  
# Рекурсивный | Матрёшка | Друг | Код | Вызов | Функция | Openmatryoshka |  
Второй вызов, как вы можете видеть, это уже рекурсивный с аргументом на единицу меньше, чем предыдущий, то есть с тройкой. Этот вызов, обратите внимание, происходит вот здесь на 8 строке кода. Он рекурсивный. На строку номер 5 до 6 мы не попадем, поскольку базовый случай размер матрешки единицы еще не достигнут. Также мы с вами не попадем и на 9 строчку, поскольку при рекурсивном вызове на 8 строчке мы снова попадем в начало функции. Третий вызов является рекурсивным, но уже с параметром 2 на единицу меньше, чем предыдущий. Ну и четвертый вызов все так же рекурсивный, его аргумент равен единице. Вызов функции с аргументом 1 приведет к следующему. Мы попадаем в тело условного оператора if. На экран выводится соответствующее сообщение о том, что достигнута самая маленькая матрешка и выполняется инструкция return на шестой строке кода. Друзья, это как раз-таки тот момент, когда череда рекурсивных вызовов останавливается. Но что будет происходить дальше? Предлагаю подумать над этим вопросом пару минут самостоятельно, а затем мы обсудим ваши догадки. you you you you На этом моменте начинается самое интересное. Предлагаю собраться с силами и внимательно проследить за следующими этапами. Мы достигли инструкции Return, на которой череда вызовов останавливается. То есть мы находимся вот здесь, сразу же после вывода сообщения. После выполнения Return мы должны вернуться в то место, откуда нас вызвали. А это, друзья, как раз-таки предыдущий шаг с номером 4, на котором отмечен вызов функции с аргументом равным единице. Мы окажемся сразу после вот этой инструкции вызова. То есть после нее давайте посмотрим, что идет. Вывод на экран. Консоль в WriteLine. Для ясности покажу суть этого возврата на простом примере. Здесь функция выводит на экран некоторое сообщение и сразу же завершается. Давайте перейдем к моменту ее вызова. Шестая строка. Вызов функцию ShowMessage. Возникает вопрос. Куда мы попадем после ее завершения? Конечно же, мы попадаем сразу же на следующую строку после вызова функции. То есть это строка номер 7, где у нас располагается некоторый вывод на экран. То же самое произойдет и при рекурсивном вызове. После вызова функции openMatryoshka находится вывод на экран. Вот здесь строка номер 9. Она и будет выполнена с аргументом, который в данный момент равен 2. То есть вот эта строчка 9 будет выполнена. На схеме она отмечена вот здесь. Обращаю внимание, друзья, size переменная равна 2. Это мы увидим на экране в консоли. Вот здесь вот в левой части экрана.  
  
# Заключение  
Далее инструкция у нас заканчивается и функция завершается. То есть после 9 строки кода функция у нас прекращает свою работу. Но не забываем, что эта функция была вызвана ранее, на шаге номер 3. Вот здесь. И мы точно таким же образом вернемся туда вместо ее вызова вот по этой стрелке несмотря на то что здесь явно не прописан оператор return после девятой строчки он будет выполнен неявно и произойдет возврат после вызова функции open матрешка с аргументом 2 идет инструкция вывода на консоль на На схеме она изображена здесь. Она выведет переменную size, значение которой равно 3. Мы это увидим на экране. На этом инструкции функции закончатся. И мы таким же образом отправимся на место предыдущего вызова. Здесь выполняются те же действия, что и ранее. На экране мы с вами увидим об открытии матрешки с размером 4. После этого мы вернемся на место первоначального вызова функции. То есть вот по этой стрелке мы возвращаемся на первоначальный вызов. И на этом, друзья, рекурсия завершится. Друзья, для понимания давайте отметим себе несколько важных моментов, которые происходят при рекурсе. Во-первых, процесс возврата, который мы видели на схеме к местам вызовов, называется раскручиванием рекурсии. Во-вторых, в процессе раскручивания рекурсии мы как бы возвращаемся во времени назад к местам вызова функции. Отметьте себе, что места вызовов сохраняются в памяти, чтобы потом можно было к ним вернуться.  
  
# Рекурсия | Вызов | Рекурсивный | Матрёшка | Друг | Работа | Функция |  
То есть в памяти будет сохранен вызов первый, второй, третий. И потом в эти сохраненные места мы будем уже делать возврат. То есть места вызова функции с аргументами 4, 3, 2, 1 будут сохранены в памяти. И затем начнется \*раскручивание рекурсии\*. То есть проход по вот этим стрелочкам, которые у нас изображены на схеме. В-третьих, помимо места, куда нужно вернуться, это у нас строка номер 8, при раскручивании, в памяти сохраняются значения параметров, с которыми была вызвана функция. То есть 4, 3, 2, 1. Делается для того, чтобы при выводе на консоль строка номер 9 выводилось корректное значение матрешки. То есть размер 2, 3, 4. Если это не сохранять, то размер у нас будет всегда один и тот же. Давайте отметим себе, что область памяти, куда сохраняется вся эта информация, называется стэком. Резюмируем сказанное и отметим основные особенности. Под раскручиванием рекурсии мы понимаем возвраты к местам вызова функций. При вызове функции в памяти сохраняются место возврата, то есть куда нам возвращаться при раскручивании, а также значение параметров при вызове. И эта область памяти называется стэк. Мы с вами рассмотрели механизм работы рекурсии на примере открытия набора матрешек на некоторой вложности. Но обратили ли вы внимание на некоторую странность, которая происходит при открытии? Если рекурсивный вызов располагается до основной работы, то матрешки у нас как бы открываются изнутри. То есть от самой маленькой до самой большой. Посмотрим еще раз на вывод. Видно, что открытие происходит от самой маленькой к самой большой матрешке Что в действительности практически нереализуемо И в целом звучит странно Мы привыкли, что матрешка открывается в обратном порядке То есть сначала самая большая и до самой вложенной Друзья, здесь дело в том, что расположение рекурсивного вызова влияет на \*последовательность действий\* в алгоритме.  
  
# Рекурсия | Рекурсивный | Алгоритм | Друг | Работа | Вызов |  
Некоторые алгоритмы могут корректно работать с расположением рекурсивного вызова как до, так и после основной работы. А некоторые алгоритмы, в частности, как наш пример, могут от этого пострадать. Поэтому рекомендую внимательно относиться к месту размещения рекурсивного вызова. Он будет влиять, от него будет зависеть последовательность операций в алгоритме. И отметьте себе важный момент, что раскручивание рекурсии будет происходить независимо от того, где расположен рекурсивный вызов. Будь то он расположен после основной работы, как в примере слева, или же он находится до запуска основных действий. Сам принцип работы рекурсии от этого не изменится. Друзья, задача с открытием матрешек вполне неплохой пример для демонстрации работы рекурсии. Как в случае нормального поведения, то есть открытия от больше и к меньше, так и в случае открытия матрешек как бы изнутри. Но хотелось бы выяснить, в каких задачах рекурсия реально полезна и упрощает их решение.  
  
# Рассмотреть | Рекурсия | Код |  
Давайте рассмотрим несколько примеров рекурсии, где ее применение действительно себя оправдывает. На экране представлен пример структуры данных, которая носит название дерево. Дерево – это один из способов хранения и упорядочивания данных, наряду с массивом. Деревья используются для хранения данных в иерархическом порядке, для поисковых задач и для сортировок. Рекурсия здесь применяется для наиболее компактной и понятной реализации обхода его элементов. Под обходом мы понимаем обеспечение доступа или же проход по всем элементам дерева. Похожую задачу мы с вами решали с массивами. Если же использовать итеративную версию обхода такой структуры данных, то она будет значительно больше по объему кода. И, кроме того, потребует использования дополнительных структур данных, таких как стэк или очередь.  
  
# Факториал | Алгоритм | Рекурсия | Работа |  
Второй пример – это алгоритм быстрой сортировки, также его называют Quick Sort. Является одним из самых быстрых алгоритмов. Превосходит по скорости пузырьковую сортировку, сортировку выбором, вставками и многие другие. Рекурсия здесь является одним из основных компонентов. Именно за счет нее и достигается высокая скорость работы. Ну и третий пример это обход каталогов файловой системы. Здесь \*рекурсивные алгоритмы\* используются для поисков файлов, папок, а также для формирования листингов содержимого. Язык C-S C Sharp и платформа.NET представляют достаточно удобные методы получения списка файлов. Однако, если нам потребуется получить список файлов на всю глубину каталогов, то есть на всю глубину каталогов урок 1, урок 2 или вложенных, то рекурсия здесь будет как нельзя кстати. Она позволит организовать этот алгоритм просто и компактно. Предлагаю немного попартиковаться и решить классическую задачу на вычисление факториала с использованием рекурсии. Для начала давайте разберемся, что такое \*факториал\* числа. Факториалом натурального числа n называется произведение чисел от 1 и до самого числа n включительно. Чтобы вычислить факториал числа 5, необходимо единицу умножить на следующее число, то есть на 2. Полученный результат необходимо умножить на следующее число, после двойки, то есть на число 3.  
  
# Факториал | Рекурсия | Рекурсивный |  
И так далее, пока мы не дойдем до самого числа. Для обозначения этой операции мы используем восклицательный знак. Отметим, что эта функция,ториал растет крайне быстро. О чем речь? При небольшом увеличении числа n, то есть то число, от которого считается факториал, значение факториала увеличивается весьма существенно. То есть факториал от 3 равен 6, от 5 он будет равен 120, но при увеличении до 10 мы уже получаем число порядка 3 миллионов. Факториал от 1 у нас равен 1, и факториал 0 равен также 1 по определению. Такое значение примето математиками. Перед тем, как начать писать код, а на первых порах может быть вообще непонятно с чего начать, как подступиться к написанию, предлагаю рассмотреть основные компоненты рекурсии для этой задачи. Первый элемент это рекурсивный вызов. Напомню, он нужен для того, чтобы решить более простую задачу и приблизиться к тривиальному случаю. Самый очевидный решить более простую задачу и приблизиться к тривиальному случаю. Самый очевидный способ получить более простую задачу, если мы говорим про факториал, то это вычислить его от меньшего числа, чем исходное. То есть, например, уменьшить его на единицу. Тем самым задача будет упрощена. Следующее, над чем нам нужно подумать, это как использовать решение более простой задачи в рекурсивном вызове. Отмечу, что эта задача может быть непростой. Представим формулу факториала несколько иначе. А именно, переставим сомножители наоборот. То есть от большего до числу к наименьшему. Ну и помним, что перестановка сомножителей на результат не влияет, как и при сложении. А теперь давайте внимательно посмотрим на формулу и подумаем, можно ли в ней выделить наиболее простую задачу. То есть можно ли в этой формуле использовать формулу, которую мы написали ранее. n-1 факториал. Заметьте, что последние n-1 сомножители, то есть начиная от n-1, n-1 факториал.  
  
# Факториал |  
Заметьте, что последние n-1 со множителей, то есть начиная от n-1, n-2 и так далее до единицы, как раз таки составляют формулу для решения более простой задачи. Итоговая формула примет следующий вид. В нее войдет решение более простой задачи, которое мы вывели ранее. И важный момент. В этой формуле также используется функция факториал. То есть формула задается рекурсивно. А это как раз-таки то, что нам нужно.  
  
# Рекурсия |  
Второй компонент – это \*условия остановки рекурсии\*. Напомню, рекурсивные вызовы нужно остановить, как только решаемая задача станет тривиальной. Или же мы подойдем к базовому случаю. Давайте посмотрим, например, вычисление факториала от числа 3 с применением рекурсии.  
  
# Факториал |  
Итак, более простая задача в этом примере это факториал от 2. В итоге мы имеем формулу как 3 умножить на факториал 2.  
  
# Заключение  
Теперь рассмотрим задачу факториал от числа 2. Как можно ее представить с применением рекурсии? 2 факториал представляется как 2 умножить на факториал от единицы. Как раз таки факториал от числа 1 и будет тривиальная задача. Это является базовым случаем, который решать дальше с применением рекурсии уже нет смысла. По определению, помните, 1 факториал равен единице. На этом, друзья, с основными компонентами, которые составляют рекурсию, все. Предлагаю оформить решение на C-Sharp. Работаем мы сегодня в каталоге урок 5. Создадим проект под каталоги с первым заданием.  
  
# Факториал |  
Итак, перейдем в файл с исходным кодом. Функция будет возвращать целое число, значение факториал. Поэтому сразу укажем тип возвращаемого значения как int.  
  
# Функция |  
Далее дадим название функции, допустим, факт. И укажем ее параметр, который будет также целым числом. Именно от него и будем вычислять значение факториала.  
  
# Параметр | Рекурсия | Рекурсивный | Видеть | Факториал | Друг | Функция |  
Далее давайте оформим рекурсивный вызов. Вернем в качестве значения функции, согласно сформированной нами формуле, n умножить на факториал от n-1. Ну а, друзья, конечно же, помним, что C-sharp в таком виде формулу факториала, конечно же, не приемлем. Для него восклицательный знак является несколько другой операцией. Поэтому напишем корректно и укажем именно функцию факт. То есть мы вычислим факториал от параметра, уменьшенного на единицу. Попробуем запустить этот код, не используя условия для остановки рекурсии. То есть укажем имя \*функции факт\* и допустим передадим ей какой-нибудь параметр, пускай будет 5. Сохраним изменения и стартанем код. Что мы видим на экране? Давайте посмотрим. Нам среда выполнения написала сообщение stack overflow. Это, друзья, значит, что область памяти, которая именуется стэком, помните, мы про него говорили, переполнилась, поскольку мы никаким образом с вами рекурсию не остановили. Условий мы для остановки не использовали. Вопрос на засыпку. А как вы думаете, какими данными заполнился стэк в нашей задаче? Подскажу. Эти данные мы обсуждали, когда разбирали работу рекурсии на примере с матрешками. Предлагаю в течение минуты подумать над этим вопросом и свои предложения написать в чат. you Да, вы были совершенно правы, если вспомнили про место возврата функции, а также значение параметров при рекурсивном вызове. В стэк помещаются именно эти данные. Если запустить бесконечную рекурсию, то стэк достаточно быстро переполнится. В принципе, что мы увидели на экране.  
  
# Рекурсия | Рекурсивный | Факториал | Функция | Работа | Код |  
Итак, добавим условия для остановки рекурсии. Напомню, это случай, когда текущая задача стала тривиальной или мы приблизились к базовому случаю. Но как в этой функции нам понять, что текущая задача тривиальна? Как мы говорили ранее, в большинстве случаев на это указывает параметр функции. Давайте проверим этот параметр на равенство единицы, поскольку при формировании этапов с основных частей рекурсии мы это отметили. Если параметр будет равен единице, мы рекурсию остановим и вернем единицу, поскольку по определению факториал единицы равен единице. В остальные уже случаи, если у нас условие n равно единице не выполняется, будет производиться рекурсивный вызов. Запустим код еще раз. Предлагаю добавить вывод полученного значения на экран, чтобы можно было проверить правильность работы. Сохраним изменения и запустим еще раз. Ну и также, конечно, не забываем о форматировании кода.  
  
# Рекурсия | Видеть | Факториал | Друг | Работа | Вызов |  
Либо контекстное \*меню Format\* Document, либо сочетание клавиш Shift Alt F. Итак, давайте посмотрим. На экране мы видим 120, что говорит о правильной работе рекурсии. Также можно проверить, правильно ли работает рекурсия при нуле. Давайте посмотрим. При нуле у нас должна рекурсия завершаться единицей, то есть возвращаться значение единицы. Но мы видим стэк слова переполняется, поскольку у нас это условие не отработало. Предлагаю это требование учесть, эту особенность, и в проверке просто добавить. Если у нас параметр равен единице или, подскажу, друзья, такой оператор или, который служит для объединения условий. Параметр равен 0, то тогда вернуть единицу. Сохраним и попробуем запустить еще раз. Да, действительно, теперь у нас факториал 0 равен 1. Друзья, отмечу, что использование рекурсии для вычисления факториала на практике не используется. Рекурсивные способы решения подобных задач мы используем исключительно для демонстрации и изучения работы рекурсии. Факториал имеет более простое решение с применением цикла. В данном случае демонстрируется цикл for. Напомню, for мы используем, когда количество итераций у нас заранее известно. Кроме того, итеративное решение не опасно переполнением стека, которое может возникать при рекурсивных вызовах. В качестве альтернативы рекурсии здесь создается цикл на n-1 итерации, то есть от 2 и до n включительным, где происходит умножение \*переменной result\* на очередное число от 2 до n. И после окончания работы цикла результат возвращается на место вызова. Мы познакомились с рекурсией в контексте языка C-sharp. Предлагаю посмотреть, как она будет выглядеть в других языках. Посмотреть, в чем будет отличие, а какие моменты будут схожи.  
  
# Компилятор | Параметр | Рекурсия | Отладка | Интерпретатор | Рассмотреть | Python | Рекурсивный | Видеть | Javascript | Матрёшка | Друг | Функция | Ошибка | Баг | C++ |  
Рассмотрим рекурсивную функцию для открытия матрешки. В языке C++ отличия минимальны. И они находятся на строке с выводом сообщения. Здесь используется конструкция std cout, которая свойственна именно языку C++. C-sharp представитель C-подобных языков. Он также имеет статическую типизацию и схожий синтаксис. В \*JavaScript\* отличий несколько больше. Во-первых, это также вывод сообщения. Используется конструкция console.log. Во-вторых, динамическая типизация. Обратите внимание, что типы возвращаемого значения у функции, а также тип параметра не указываются. Вместо этого мы видим ключевое слово function, что говорит интерпретатору JavaScript о том, что создается функция. Кстати, вопрос на засыпку. Все ли помнят, что такое \*интерпретатор\* и \*компилятор\*? Ну и третий пример – это \*Python\*. Здесь уже видны существенные отличия в синтезе и оформлении блоков. Фигурных скобок здесь уже не используется. В отсутствии указания типа данных возвращаемое значение функции, тип данных у параметра а также при выводе на экран здесь используется функция print с отличиями все понятно давайте посмотрим на общие части во всех этих трех примеров не трудно видеть что все составные части рекурсии в том или ином виде да присутствуют во всех трех примерах будь то компилируемые в языке C++, будь то интерпретируемые в JavaScript и Python. Что мы имеем? Основную работу – это вывод, сообщение об открытии матрешки, проверка условия, не достигли ли мы базового случая, также присутствует во всех трех языках. Ну и, конечно же, рекурсивный вызов, если размер матрешки позволяет ее открывать. Друзья, отвлечемся от рекурсии и поговорим о такой тематике, как отладка. Посмотрим, что она из себя представляет, какие разновидности имеются, а также попробуем применить несколько методик отладки на наших задачах. Под отладкой мы понимаем процесс поиска и исправление ошибок в коде. То есть отладку мы можем использовать, если у нас есть подозрение, что в коде есть ошибки или, другими словами, \*баги\*. Кроме того, отладка может быть применена, если не совсем понятно или же совсем непонятно, как работает программа. Отладку можно проводить путем вывода интересующей информации на консоль. Или, например, в файл. Также можно пользоваться специальными инструментами среды разработчиков.  
  
# Рассмотреть |  
Давайте рассмотрим каждый из этих способов подробнее.  
  
# Параметр | Рекурсия | Видеть | Факториал | Работа | Друг | Функция | Вызов | Стек | Код |  
Итак, первый способ – это \*вывод информации\* о ходе работы программы на консоль. Или, допустим, в файл. Возникает вопрос, а какая информация может быть выведена? Можно выводить значение переменных, значение параметров функции, а также просто сообщение о том, что выполняется та или иная строка кода. Последний способ при выводе сообщений позволяет понять попали ли мы в конкретный участок кода или нет. Давайте попробуем применить этот способ отладки на нашей задаче. Вернемся к задаче с факториалом и давайте закомментируем участок, который отвечает за остановку рекурсии. Напомню, что при запуске функции без остановки рекурсии мы получим ошибку переполнения \*стек\*. Давайте еще раз на всякий случай проверим. Допустим, да, факториал от 5. Видим, да, что происходит ошибка. После выполнения функции, то есть попытки запуске порядка 24 тысяч раз стек переполнился. Чтобы посмотреть, что происходит при работе этой функции, давайте добавим отталочный вывод на консоль. Выведем значение параметра в начале функции. Допустим, вот здесь напишем консол.writeLine. Ну и, соответственно, сам параметр. Он равен у нас n. Сохраним изменения и запустим код еще раз. Видно, как на экран вводится значение параметра, и каждый раз он уменьшается. Обратите внимание, что параметр при выводе, который мы видели ранее, не остановился на значении 0 или 1. Он уменьшался до тех пор, пока у нас не произошла ошибка, сообщение которой мы, в принципе, видим на экране. Как раз для этого мы с вами используем остановку рекурсии, чтобы избежать бесконечного числа вызовов. Еще раз, чтобы проследить, запустим код. И я остановлю нажатием клавиш Ctrl-C. То есть мы видим, что количество вызовов уже ушло за 6000 и продолжает расти. Вернем остановку рекурсии и давайте запустим код еще раз. Раскомментируем этот фрагмент, сохраним и запустим. На экране видим последовательность значений параметров от 5 до 2. При значении параметра равном единице, согласно коду, в принципе, мы это видим при выводе, рекурсия завершается. И мы видим ответ, который мы пишем на строке номер 11, это 120. Чтобы понять, да, именно что мы попали вот сюда, в условие if, да, в ветку до условия if, давайте здесь добавим также отладочный вывод на экран. Ну, допустим, напишем следующее, что рекурсия остановилась. Стоп. И укажем значение параметра. И не забываем указать, что это не просто строка, а строка с форматированием. Или же более правильный термин – интерполяция строки. Сохраним наработки. Запустим еще раз. В конце мы видим попадание в ветку условной конструкции if. И, соответственно, после этого видим ответ. А теперь вопрос, друзья. А как проследить за процессом раскручивания рекурсий? Сейчас мы с вами пронаблюдали процесс вызовов. Но когда начинается череда возвратов после рекурсивных вызовов, этот момент также хотелось бы увидеть на экране. Предлагаю подумать над этим моментом самостоятельно и оставляю его в качестве домашнего задания. То есть необходимо добавить в код некоторый отладочный вывод на правильные места, который позволит увидеть, как раскручивается рекурсия, то есть как формируется значение факториала.  
  
# Параметр | Рекурсия | Факториал | Функция | Вызов |  
Сначала равно единице, затем двойки, тройки, шестерки и так далее. То есть постепенно эти умножения. Или же другими словами процесс возвратов к местам вызовов. Вывод на экран может иметь следующий вид, когда вы уже примените отладочный вывод. Желательно увидеть значение параметра функции при раскручивании рекурсии. То есть параметр вот он 2, 3, 4, 5. А также вычисленное значение факториала на каждом шаге. Как можно видеть из примера, факториал считается постепенным. На каждом возврате к месту вызова он у нас увеличивается.  
  
# Абзац  
От 1 и так далее до 120. Дам небольшую подсказку. Скорее всего, вычисление факториала на строке 9 можно разделить на составные части. То есть вот здесь, когда мы возвращаем ответ. Это позволит встроить отладочный вывод на нужную позицию.  
  
# Рассмотреть | Отладчик |  
Двигаемся далее и рассмотрим более продвинутый способ отладки. Отладки – это использование специального инструмента или же отладчика. Отметим себе, что \*отладчик\* – это программа, которая позволяет запустить программу в особом режиме, который так и называется режим отладки. Обычно этот инструмент встроен в среду разработки, но может устанавливаться и отдельно. В редакторе \*VS Code\* нам понадобится расширение \*C-Sharp Development Kit\*, чтобы воспользоваться отладчиком. Отладчик позволяет запускать код построчно, вручную переходя от инструкции к инструкции.  
  
# Работа | Отладчик | Код |  
Мы можем вручную выполнять первую строку, затем вторую, третью, четвертую и так далее, пока не дойдем до конца программы. С его помощью можно выполнять циклы, контролируя каждую итерацию. Первая итерация, вторая, третья. То есть выполнение тела цикла. Мы можем приостановить программу во время ее выполнения на указанной строке. Это указано в примере на шестой строчке. То есть отладчик здесь приостановил выполнение кода и ожидает команды от программиста. В ходе работы также можно просматривать значения переменных и массивов. Для этого используются отдельные окна, отображение которых можно включать в настройках. Здесь мы наблюдаем переменную n, ее значение наравно 5, переменную i, у нее нулевое значение, ну и здесь я немного подскажу, это массив с аргументами программы. В данном случае он у нас нулевой. Мы аргументы на вход программы никакие не передаем. Давайте вернемся к задаче с факториалом и попробуем запустить программу в режиме отладки. Для запуска отладчика нам потребуется установить расширение C-Sharp DevKit. Давайте посмотрим, как он выглядит. У меня он уже установлен. Это официальное расширение от Microsoft, которое и содержит в себе инструменты отладки.  
  
# Абзац  
Итак, перейдем в код. После установки отладчика в правом верхнем углу становится доступна кнопка для его запуска.  
  
# Код |  
Итак, давайте нажмем «Стартануть отладку» проекта, который ассоциирован с этим файлом исходного кода.  
  
# Отладчик | Видеть |  
Итак, мы видим, что отладчик стартует. Сообщения желтым цветом говорят о инициализации запуске отладчика. Мы видим, что программа отработала. Видим полностью вывод на экран с ее сообщениями. Однако выполнение программы не остановилось. Она полностью отработала и завершилась. Дело в том, что мы не указали отладчику, а на какой строчке ему нужно остановить выполнение.  
  
# Отладчик | Друг |  
То есть, допустим, на 5, на 7, на 10 и так далее. Возможно, некоторые из вас замечали, что слева от номеров строк можно ставить красные точки. Так вот, друзья, это и есть точки остановки программы, которую использует отладчик. То есть, когда отладчик запускается, он обращает внимание на эти точки и как бы приостанавливает работу программы в тех местах, где вы ему указали. Попадая на строчку, допустим, на эту с красной точкой, выполнение приостанавливается, и отладчик будет ожидать команды от программиста.  
  
# Параметр |  
Итак, давайте поставим точку останова на строке номер 10, там где выводится очередное значение параметра. И таким же образом запустим отладку.  
  
# Параметр | Видеть | Факториал | Отладчик | Функция | Вызов |  
Итак, отладчик запускается, мы видим, что \*запустилась функция\* и выполнился код до поставленной точки. Строка, которая подсвечена желтым, где здесь есть желтый индикатор, еще не выполнилась. Отладчик в данный момент ожидает от нас команду. Обратите внимание на панель управления отладкой в верхней части экрана. Нам здесь наиболее интересны кнопки. Продолжить от английского continue, клавиши F5. Сделать шаг с обходом, клавиши F10, step over. Сделать шаг с заходом, клавиши F11. А также остановить отладку, shift F5. Давайте отметим себе, можно даже в комментариях, что кнопка шаг с заходом, то есть F11, выполняет инструкцию, на которой находится курсор и переходит к следующей. Соответственно, напишем себе шаг с заходом. Откуда такое название? Шаг с заходом. Дело в том, что если на пути отладчика встречается функция, то отладчик зайдет в тело этой функции и будет уже идти по ней. Кнопка шаг с обходом также выполняет инструкцию, на которой находится курсор и переходит к следующей. Давайте себе это отметим. F10 это у нас шаг с обходом. С обходом тоже имеет свой смысл выражение. Это значит, что если на пути отладчика встречается вызов функции, допустим, факт, то отладчик в эту функцию заходить не будет, а перейдет сразу к следующей инструкции. То есть, допустим, если бы здесь была бы какая-то инструкция сразу же после вызова функции факт, то он перешел бы к этой инструкции. Давайте нажмем F10 и перейдем к следующей инструкции. Видно, как выполнение перешло на следующую строку. Индикатор загорелся на 11 строке. И давайте посмотрим на консоль. Здесь появился вывод параметра функции 5. То есть отработала функция вывода параметра на экран. Текущая строка номер 11 содержит вызов функции. Факториал от n-1. Давайте нажмем клавишу F11, чтобы отладчик зашел в эту функцию. Мы снова оказались в функции факториал, поскольку вызов был рекурсивным.  
  
# Параметр | Рекурсия | Функция | Работа | Вызов |  
Нажатием клавиши F10 можно выполнить инструкции далее, вплоть до рекурсивного вызова. Давайте это проверим. Нажимаем F10, выполняем проверку условной конструкции. Внутрь условия мы не попадаем и отправляемся на строчку номер 10. Нажимаем F10 еще раз и видите на консоли появляется значение параметра уже 4, на единицу меньше чем предыдущие. Отметим себе, что клавиша F5 может использоваться для команды продолженная. Это значит, чтобы перейти к выполнению инструкции, на которой установлена точка останова. То есть мы перейдем вот на эту инструкцию при нажатии клавиши F5. Давайте нажмем клавишу F5 и мы снова попадаем на десятую строку. То есть мы зашли в функцию factorial, сделали проверку в условной конструкции и снова оказались на точке остановы, которая была поставлена. Таким образом мы можем несколько раз нажать клавишу F5 и поочередно увидеть параметры при рекурсивных вызовах. При некотором нажатии начнется раскручивание рекурсии, которая завершит в целом алгоритм. Также отмечу, что \*точки останова\* могут добавляться во время работы отладчика. Можно поставить точку в начале функции и запустить отладку. Давайте попробуем это сделать.  
  
# Видеть | Отладчик | Функция | Код | Друг |  
Итак, поставим точку в самом начале функции, запустим отладку. Мы оказались на первой строке функции факт и прямо в режиме отладки можно добавить точку на произвольную строку, допустим на 11. Для того чтобы переместиться сразу на 11 строчку кода, то есть выполнить все что было до, мы нажимаем клавишу F5, то есть продолжить. Это значит у нас до следующей точки. у нас до следующей точки. Нажимаем F5 и оказываемся на строке 11, там где стоит точка. На консоли мы также видим отладочный вывод. Отладчик останавливается в ожидании следующей команды. Таким образом, нажимая клавишу F5, мы перемещаемся между двумя этими точками останова, тем самым постепенно выполняя алгоритм. Друзья, при отладке вы, наверное, обратили внимание на окно слева. Оно содержит имена переменных и значения их в определенный момент отладки. Обратите внимание, что при продвижении по алгоритму, я нажимаю клавишу F5, значение переменная n уменьшается от 5 до и стремится к 1. После чего выполнение будет завершено. Как мы говорили ранее, это достаточно удобный инструмент отлаживания циклов, когда нужно следить за значениями счетчиков и другими переменными.  
  
# Работа | Отладчик | Друг | Код |  
Итак, друзья, давайте резюмируем основные моменты по отладке. Основное назначение отладки – это поиск ошибок в коде. Основное назначение отладки – это поиск ошибок в коде. Также отладку можно использовать, если не совсем понятно, как программа работает. Самый простой способ отладки – это \*отладочная печать\*. То есть вывод информации в определенных местах кода. Мы можем выводить значения переменных, элементы массива или какие-либо сообщения. Отладочная печать позволяет нам понять, допустим, запустилась функция или нет, на какой строчке произошла ошибка, а также запускается ли программа вообще. Более продвинутый способ отладки – это использование специального инструмента, который называется отладчик. Отладчик позволяет нам запустить программу построечно и отслеживая значение переменных, как мы это делали ранее, на каждом участке. Однако стоит заметить, что использование отладчика не всегда возможно. Связано это с тем, что программа может запускаться каким-либо нестандартным образом или запускается в условиях, в окружении, где запуск отладчика попросту невозможен. В таких случаях я бы предложил использовать отладочный вывод в файл. Запускаем программу и направляем сообщение в файл. Затем мы анализируем этот файл либо по окончании работы программы, либо же во время.  
  
# Рассмотреть | Код |  
Следующая тематика, которую мы рассмотрим, это принципы разработки программного обеспечения. Некоторые из них мы так или иначе использовали в наших задачах. Предлагаю их резюмировать. На экране представлены достаточно популярные принципы, которые позволяют улучшать структуру кода и бороться с его сложностью. Первый принцип говорит нам о том, чтобы не дублировать фрагменты кода, которые уже написаны. То есть от английского он переводится как «не повторяй себя» или покороче «не повторяйся».  
  
# Абзац  
Второй принцип призывает не усложнять код без необходимости, сохраняя его простым, собственно, как здесь и написано. То есть \*keep it simple\*,раняйте простым. Максимально простым. Третий принцип. Предлагает отказаться от функциональности, которая в продукте не требуется. Расшифровывается, как вам это не потребуется.  
  
# Рассмотреть | Функция | Код |  
Рассмотрим принципы подробнее и приведем примеры их нарушения. Некоторые из примеров сгенерировал я, а некоторые взял из работ студентов на текущем курсе. Первый принцип рекомендует избегать дублирования одинаковых фрагментов кода. Мы с вами им неоднократно пользовались, когда разрабатывали функции. Пример. Использование функции для вывода массива на экран. Вместо того, чтобы писать цикл для вывода массива несколько раз, мы вызывали соответствующую функцию. Почему дублирование кода это пустая трата времени? Дело в том, что вам придется будет поддерживать логику программы, а также тестировать код сразу в двух местах. Причем, если код изменится в одном месте, то вам необходимо его будет править и во втором. В большинстве случаев дублирование кода происходит из-за незнания особенностей системы. То есть мы не знаем, что какие-то компоненты уже есть, а также, что они могут отсутствовать. Как здесь сказано, прежде чем что-либо писать, необходимо осмотреться и подумать. Возможно, эта функция уже реализована. Возможно, эта бизнес-логика присутствует в каком-то другом месте. Повторное использование функции – это хорошее решение данной проблемы.  
  
# Абзац  
Одно из самых простых способов – это введение в программу функций.  
  
# Рассмотреть | Код |  
Рассмотрим пример. Это решение одного из студентов задачи с выводом на экран четных чисел. От одного и до некоторого введенного n. Вначале считывается число с консоли, затем запускается цикл, в котором перебираются числа от 1 до n и каждый раз проверяются на четность. Если это так, то число выводится на экран. Как вы думаете, есть ли здесь нарушение принципа \*do not repeat yourself\*? Свои предположения можете написать в чат. К этому примеру мы еще вернемся. Следующий принцип говорит о том, что не нужно усложнять решение задачи, когда можно обойтись более простым решением. Одна из распространенных ошибок – это использование новых инструментов исключительно из-за того, что они красивые и блестят. Некоторые программисты таким образом стремятся продемонстрировать знания технологии, даже если эти инструменты не совсем подходят для решения задач. Этот принцип подразумевает написание понятного кода. Если код непонятен, то он автоматически становится сложным для его сопровождения, как указано здесь, для понимания последующей модификации.  
  
# Рассмотреть |  
Рассмотрим такой пример. Цель программы – посчитать сумму элементов массива. В ней создается массив на 5 элементов, затем переменная для хранения суммы, и в нее помещается первый элемент.  
  
# Переменный | Stupid | Функция | Код | Друг |  
Далее организуется цикл от последнего элемента до первого, в котором осуществляется суммирование. Друзья, видите ли вы в этом решении что-нибудь странное и нетиповое? Можно ли сказать, что это решение было искусственно усложнено? Предлагаю подумать над этим вопросом самостоятельно и обсудить на семинаре. Резюмируя сказанное и принцип Keep it simple, \*stupid\* в целом, приведу цитату уже знакомого вам автора Стива Макону. Он считает управление сложностью самым важным техническим аспектом в разработке программ. Более того, управление сложностью Стив Макону считает главным техническим императивом в разработке ПУ. И на протяжении всей своей книги совершенный код к этому термину обращается. Ну и третий принцип – это отказ от избыточной и ненужной функциональности. Он говорит нам о том, чтобы реализовывать только то, что нужно в данный момент. Программист не должен добавлять в функционал программы те компоненты, которые не значатся в техническом задании. Перечислим остальные идеи, которые лежат в основе этого принципа. Не стоит писать код, который может пригодиться позже. Например, пусть стоит задача вычислить среднее арифметическое от элементов массива. После реализации такой функции по вычислению среднего арифметического мы решили еще реализовать и функцию по вычислению среднего гефметического, мы решили еще реализовать и функцию по вычислению среднего геометрического. Просто так, на всякий случай. Так вот, друзья, от этой идеи лучше отказаться, если оснований для использования этой функции среднего геометрического нет. Не стоит бояться удалять лишний код. Его всегда можно восстановить из репозитория. Конечно же, если он \*сохранен соответствующими\* коммитами. Вы, должно быть, заметили, что этот принцип Ягни, который мы рассматриваем, похож на принцип Кис. Он старается делать вещи как можно проще. Но дело в том, что принцип Кис старается искать более простые решения, в то время как принцип Ягни не делает никаких решений. Вернемся к примеру с подсчетом количества четных чисел от 1 до n. Обратите внимание на код в блоках if и else. И в первом, и во втором блоке присутствует инструкция увеличения переменной на n. Она увеличивается на единицу. Также переменная i также увеличивается на единицу. Также переменная i также увеличивается на единицу. В подобных случаях повторяющийся код выносят за пределы операторов if-else. Например, размещая его сразу после них. Давайте применим принцип do not repeat yourself. После рефакторинга код примет следующий вид. Действительно, вне зависимости от того, является ли число четным или нет, нам необходимо перейти к проверке следующего числа. Для этого мы вынесли указанную операцию за блок if else. То есть вот этот блок, переходу к следующему числу, увеличивает его на единицу. Затем его проверяют. Обратите внимание, что блок else стал пустым. Разумеется, от него можно избавиться, просто удалив его. Друзья, на этом рефакторе этого фрагмента кода не заканчивается. И я предлагаю вам подумать, а как здесь применить \*принцип Ягнида\*, то есть избавиться от избыточной функциональности. Подскажу. В качестве личных фрагментов можно рассмотреть \*переменные\*. В качестве второго примера, где я бы хотел продемонстрировать нарушение принципа Ягни, это заполнение массива случайными числами. Здесь создается массив на 10 элементов.  
  
# Код |  
Далее в цикле for он инициализируется нулями и затем, предварительно создав объекта Random, он уже инициализируется случайными числами. Как вы думаете, есть ли в этом примере \*избыточная функциональность\*? То есть операции, от которых можно было бы избавиться. Необходимости в зануление массива здесь нет. Поскольку при его создании элементам присваивается значение по умолчанию. Если речь идет о числах, то это нули. Цикл на строчках 4-7 можно полностью удалить. На поведении программы это никак не скажется. Резюмируем рассмотренные принципы на приведенной картинке. В качестве фигур могут выступать фрагменты кода, над которыми будем применять принципы избавиться от лишней функциональности, упростить код и, соответственно, избавиться от дублирования. На первом переходе мы отказываемся от избыточной функциональности, удаляя лишние фигуры.  
  
# Код |  
Второй переход упрощает оставшиеся программные сущности, применяясь принцип KeepItSimpleStupid. Ну и третий переход объединяет сущности одинаковые в единое целое. Например, одинаковые фрагменты кода могут быть заменены на вызовы функций.  
  
# Рассмотреть | Рекурсия |  
В заключение сегодняшнего занятия давайте рассмотрим несколько типовых вопросов по рекурсии, которые вы можете услышать на собеседовании.  
  
# Рассмотреть | Рекурсия |  
Так же, как и на предыдущем занятии, рассмотрим теоретические вопросы, а также вопросы, касающиеся практики. Первый вопрос вероятен на \*собеседование джуниор-программиста\*. Его цель понять, может ли программист объяснить, что такое рекурсия как в широком, так и в узком смысле. Напомню, в широком смысле рекурсия – это описание объекта или процесса в рамках этого же объекта или процесса. Если речь идет о программировании, то здесь уже понимается некий процесс, в котором функцию вызывает сама себя.  
  
# Заключение  
Второй вопрос касается основных компонентов рекурсий. Требуется их перечислить и кратко описать. \*Рекурсивная функция\* состоит из блока основной работы, рекурсивного вызова, который предназначен для решения более простой задачи, а также базового случая, который предназначен для остановки рекурсивных вызовов. В этом вопросе вас могут попросить привести пример рекурсивной функции и показать ее составляющие части. Третий вопрос проверяет, есть ли у вас понимание механизма работы рекурсии. Его основная идея – это вызов функции в самой себе для решения более простой задачи, чем текущей. С каждым новым вызовом задача становится проще. В частности, уменьшается объем вычислений. Эти вызовы называются рекурсивными и продолжаются до тех пор, пока мы не дойдем до некоторого базового случая, где рекурсивный вызов уже не требуется. После достижения базового случая начинается процесс раскручивания рекурсии. Последовательность возвратов к местам вызовов функций. Этот процесс мы подробно рассматривали на примере с матрешками. Следующий вопрос охватывает не только алгоритм работы рекурсии, но и особенности хранения данных в памяти. Вернемся к примеру с матрешками. Напомню, в процессе рекурсивных вызовов в памяти сохраняются места этих вызовов, как мы видели в коде, а также значение параметров, с которыми они вызывались, то есть от 4 до 1. Эта область памяти называется стэком. Информация о работе рекурсии, о вызовах помещается в стек по мере работы рекурсии. Сначала помещается информация о вызове с параметром 4, затем с параметром 3, 2 и последнюю очередь поместится информация о вызове с параметром 1. При раскручивании рекурсии информация о вызове будет браться именно из стека. Но, друзья, обратите внимание, обязательно в обратном порядке. То есть сначала будет браться информация о первом вызове, о втором, о третьем и о четвертом. Помните, мы с вами шли в обратном порядке по местам вызова функций. Точно так же информация извлекается тэка в обратном порядке, начиная от самого последнего вызова до самого первого. Давайте подумаем, а что произойдет, если условие для остановки рекурсии написано неправильно или же отсутствует совсем. Рекурсивные вызовы будут выполняться бесконечно, каждый раз с меньшим значением аргумента. После достижения базового случая, допустим это open матрешка с аргументом 1, вызовы продолжаются, несмотря на то, что аргумент уже уходит в отрицательные значения. Это говорит о том, что условие остановки не работает. Но самое важное, что информация о вызовах будет также помещаться в стэк. Стэк имеет фиксированный размер. Обычно он занимает порядка 1-2 мегабайт. В некоторый момент его свободное место закончится и он будет заполнен. В этом случае информацию об очередном вызове попросту будет некуда сохранить. В данном случае стэк у нас заполнился на количестве вызовов 5 и информацию о 6 вызове его уже сохранять некуда. Друзья, здесь, конечно же, \*синтезированный пример\* с стэком небольшого размера, буквально на 5 вызовов. На практике, как мы видели, число вызовов может достигать порядка десятков тысяч, после чего происходит ошибка. При совершении очередного рекурсивного вызова программа попытается сохранить в стеке информацию о нем. В данном примере это вызов с аргументом минус 1, однако места там уже свободного нет. Как раз таки здесь, друзья, мы увидим информацию о том, что стэк переполнен. Чтобы это не допустить, мы используем базовый случай, при достижении которого рекурсивные вызовы вот здесь мы останавливали. Еще раз. Под переполнением стэка мы понимаем ситуацию, при которой его свободный объем полностью исчерпан и предпринимается попытка добавить в него данные. На этом лекция подходит к концу. Давайте подведем ее итоги. Если мы говорим про рекурсию, то стоит помнить про определение в широком смысле и более узком, которое уже относится непосредственно к программированию. Напомню, что рекурсия состоит из рекурсивного вызова, основных вычислений, некого простейшего или базового случая, который служит для остановки рекурсии. Наиболее популярные области применения рекурсии – это обходы деревьев, сортировки, а также поиск файлов и подкаталогов в каких-либо директориях. Рекурсию следует применять с осторожностью, в тех случаях, когда она действительно оправдывает свое применение и упрощает алгоритмы или делает их более компактными. После блога задач на рекурсию мы с вами рассмотрели два способа отладки программ. Это с использованием отладочной печати на консоль или файл, а также с использованием особого инструмента. В заключении лекции мы обсудили принципы создания хороших программ и рассмотрели несколько возможных вопросов в собеседовании на тематику рекурсии. Ну что, друзья, на этом теоретическая часть курса заканчивается. Впереди у нас семинар. Несмотря на небольшое количество лекций, мы с командой \*Geekbrains\*, помимо базовых тематик, таких как арифметика, условные операторы, циклы, массивы, функции, постарались поместить туда и дополнительные разделы. Такие как рекомендации по написанию хорошего кода, особенности работы с текстовым типом данных, отладка программ, блок вопросов с собеседованием и несколько других. Если какие-то тематики остались непонятными, то я рекомендую ознакомиться с дополнительной литературой. Дополнительными источниками я рекомендую пользоваться для закрепления материала, а также для более глубокого его изучения. Позвольте пожелать вам успехов в дальнейшем освоении профессии, грамотных преподавателей и интересных задач. На этом, друзья, курс знакомства с языками \*программирования завершается\*. Всем большое спасибо за работу и внимание. И до новых встреч! you