1С:ЭЛЕМЕНТарно!
1С:Элемент для будущих разработчиков: практикум 10-11 класс

Книга предназначена для школьников 10-11 классов и тех, кто хочет освоить основы программирования с нуля, используя современный и доступный язык – 1С:Элемент. Подходящая как для самостоятельного изучения, так и для использования в образовательных программах, книга пошагово вводит читателя в мир программирования – от базовых понятий до прикладных задач.

Материал структурирован для новичков и тех, кто имеет начальные знания: от простых программ и базовых конструкций к сложным типам данных и современным технологиям. В издании представлены теоретические объяснения, практические задания с решениями и справочные разделы для быстрого поиска информации.

Это отличный старт для тех, кто хочет сделать первые шаги в IT-разработке.

Автор – Максим Радченко. В книге использованы примеры Елены Хрусталёвой по работе с базовыми типами «1С:Элемента».

Электронная книга в формате pdf; ISBN 978-5-9677-3574-5. Версия издания – 15 сентября 2025 г.

Электронный аналог издания "1С:ЭЛЕМЕНТарно! 1С:Элемент для будущих разработчиков: практикум 10-11 класс " (ISBN 978-5-9677-3573-8, М.: ООО "1С-Паблишинг", 2026; артикул печатной книги по прайс-листу фирмы "1С": 4601546149282; по вопросам приобретения печатных изданий издательства "1С-Паблишинг" обращайтесь к партнеру "1С", обслуживающему вашу организацию, или к другим партнерам фирмы "1С").

Содержание

Глава 1. Предисловие....4

Глава 2. Установка и начало работы....5

Глава 3. Настройка рабочего пространства....11

Глава 4. Базовый уровень....20

Базовые понятия....20

Арифметические операции....39

Операции со строками....42

Тип «ДатаВремя» и операции с датами....43

Тип «Булево» и логические операции....52

Булевы операции....55

Инструкция «если»....61

Красивый скрипт....66

Инструкция «для по»....73

Инструкция «выбор»....79

Методы....81

Область видимости имён....91

Чтение и отладка методов....96

Глава 5. Коллекции, структура, перечисление....101

Экземпляры, свойства, методы и конструкторы....101

Иерархия типов, тип «Тип»....103

Коллекции и обобщённые типы....108

Массив....110

Соответствие....130

Множество....137

Структура....140

Перечисление....148

Глава 6. Углублённый уровень....152

Составные типы....152

Работа с числами, методы чисел....157

Работа со строками, методы строк....166

Типы для работы с датой и временем....181

Регулярные выражения....187

Работа с массивами....193
Работа с соответствиями.... 199
Константы....200
Исключения....200
Работа с методами 207
Модульная разработка.... 213
Функциональные типы....216
Динамическая типизация....224
Механизм отражения.... 225
Особенности системных методов и типов....226
Ввод английских символов без переключения раскладки клавиатуры....227
Рекомендации по написанию кода....228
Глава 7. Прикладные возможности.... 245
Работа с файлами.... 245
Чтение и запись текстовых файлов....250
JSON....254
XML....269
HTTP-запросы....273
Глава 8. Список терминов.... 284
Глава 9. Список инструкций, операций и символов.... 286
Глава 10. Решения заданий.... 291
Глава 11. Выходные данные....307

Глава 1. Предисловие

Состав книги

Книга состоит из нескольких частей.

Базовый уровень *(20)* предполагает, что раньше вы не имели дела с программированием. Здесь вы овладеете базовыми понятиями «1С:Элемента» и основными операциями. В конце многих разделов содержатся задания для самостоятельного выполнения. В самом конце книги есть раздел Решения заданий *(291)*. В нём вы сможете посмотреть правильные ответы.

Раздел **Коллекции, структура, перечисление** *(101)* познакомит вас со сложными типами данных и с типами, которые вы можете создавать сами.

Углублённый уровень *(152)* рассчитан на то, что вы уже имеете некоторый уровень знаний в программировании. В этом разделе частично затрагиваются темы, которые вы уже изучали ранее, но на более глубоком уровне. Кроме этого вы познакомитесь с новыми понятиями и возможностями «1С:Элемента».

Раздел **Прикладные возможности** *(245)* посвящён работе с файлами, файловой системой, обработкой JSON, XML и выполнению HTTP-запросов.

В конце книги находятся несколько справочных разделов. Они помогут вам быстро найти то, о чём вы читали раньше.

Список терминов *(284)* содержит основные понятия, которые вводятся и объясняются на протяжении книги.

Список инструкций, операций и символов *(286)* поможет вам, когда вы смотрите в скрипт и не понимаете, что это «значит». Тут вы найдёте перечисление всех ключевых слов «1С:Элемента», всех символов, которые обозначают операции или просто используются в синтаксисе.

Благодарность

Спасибо Яну Радченко за первое чтение книги. Его отзывы и комментарии помогли улучшить эту книгу и сделать её более понятной.

Глава 2. Установка и начало работы

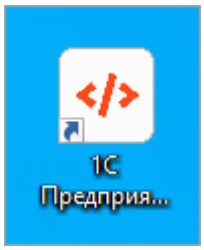
Установка

Скачать дистрибутив можно по следующему адресу: https://lang.1c.ru/

Распакуйте архив, в котором содержится дистрибутив.

Запустите **1ce-installer.exe** и нажмите **Установить**. Начнётся установка.

После установки у вас на рабочем столе появится ярлык **1С Предприятие.Элемент Скрипт IDE 1.0.0**.

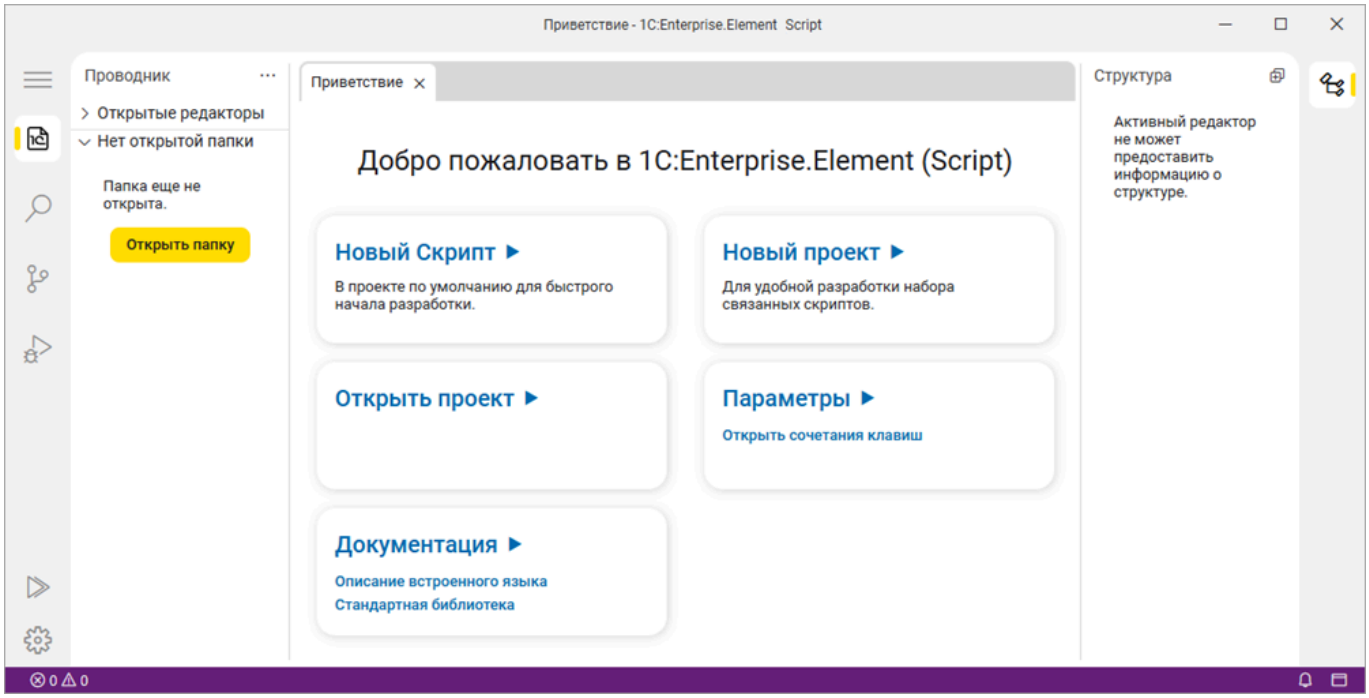


Кроме этого такой же ярлык появится в меню **Пуск**.

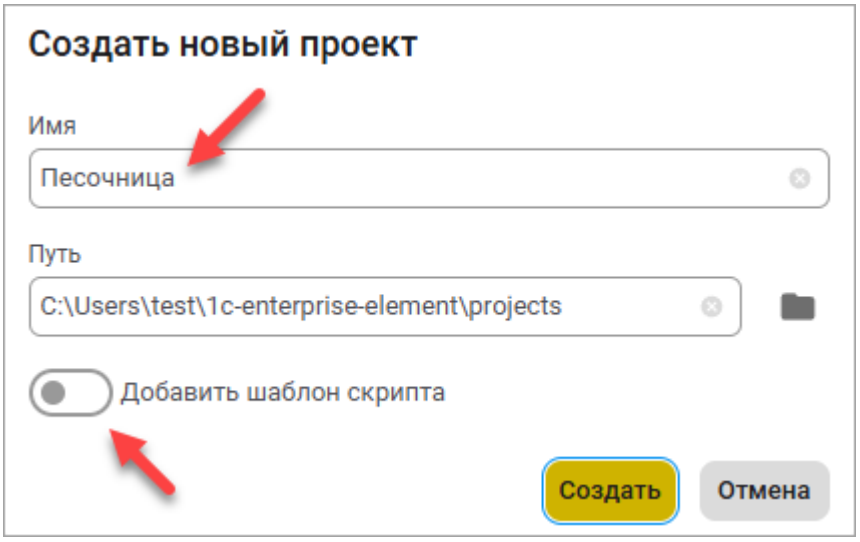
Чтобы запустить «1С:Элемент», можете использовать любой из этих двух способов, какой вам больше нравится.

Первый запуск

После первого запуска вы увидите начальный экран.

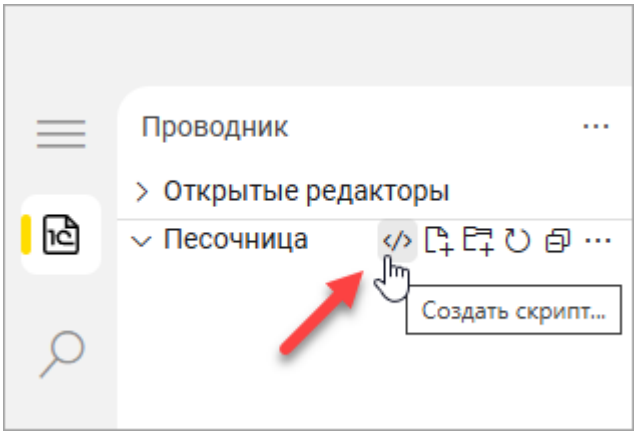


1. Нажмите **Новый проект**, укажите название проекта и выберите каталог, в котором будут находиться ваши скрипты.
   Выключите **Добавить шаблон проекта**. Это полезная функция на будущее, но сейчас первый скрипт вы создадите сами.



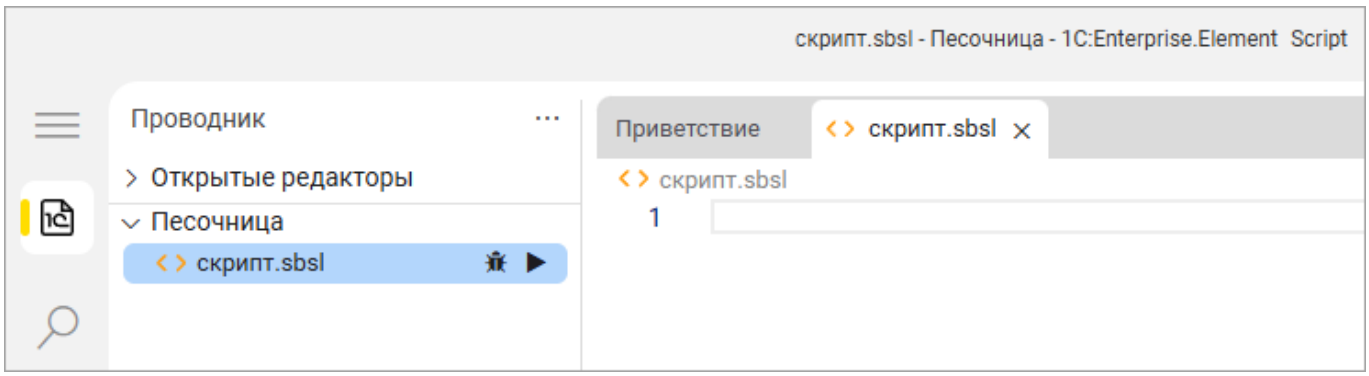
2. Среда разработки перезапустится, и в левой панели **Проводник** будет открыт ваш каталог.

3. Подведите мышь к заголовку и нажмите **Создать скрипт...**



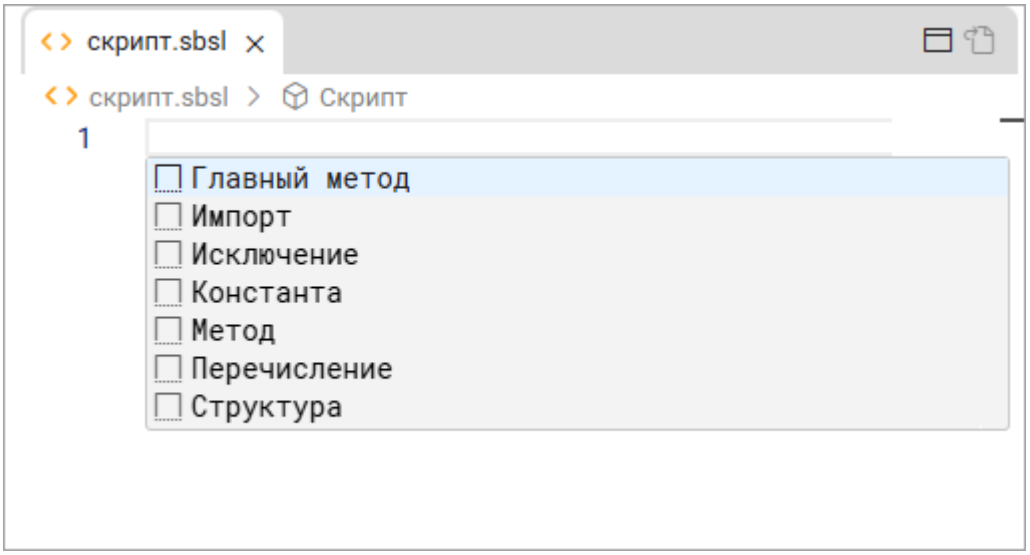
4. Введите имя файла, например **скрипт**, нажмите **ОК**.

5. Среда разработки откроет файл вашего скрипта, и вы можете писать в нём код.

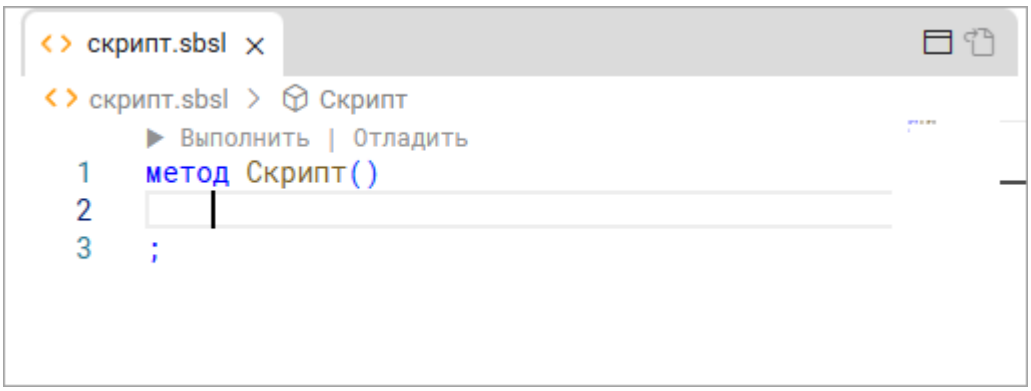


Первый скрипт

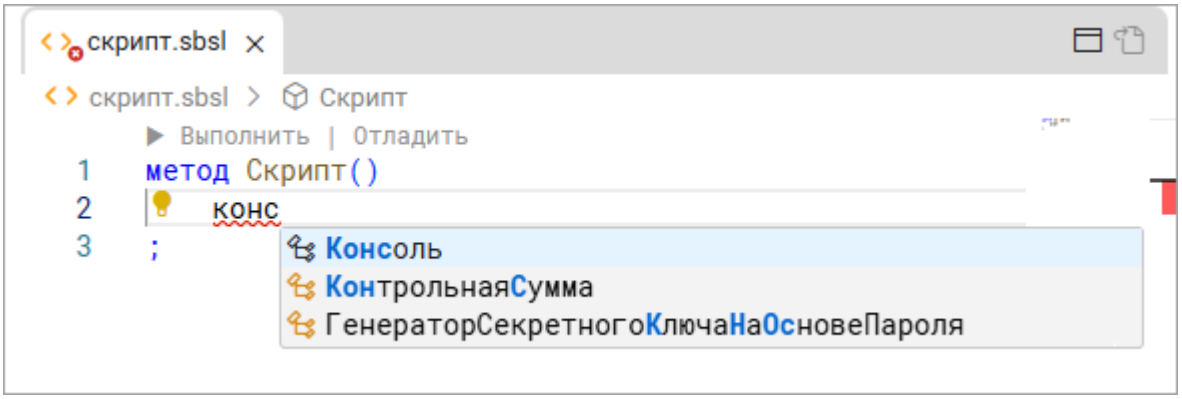
1. Установите курсор в начало первой строки и нажмите **Ctrl+Пробел**. Откроется контекстная подсказка.



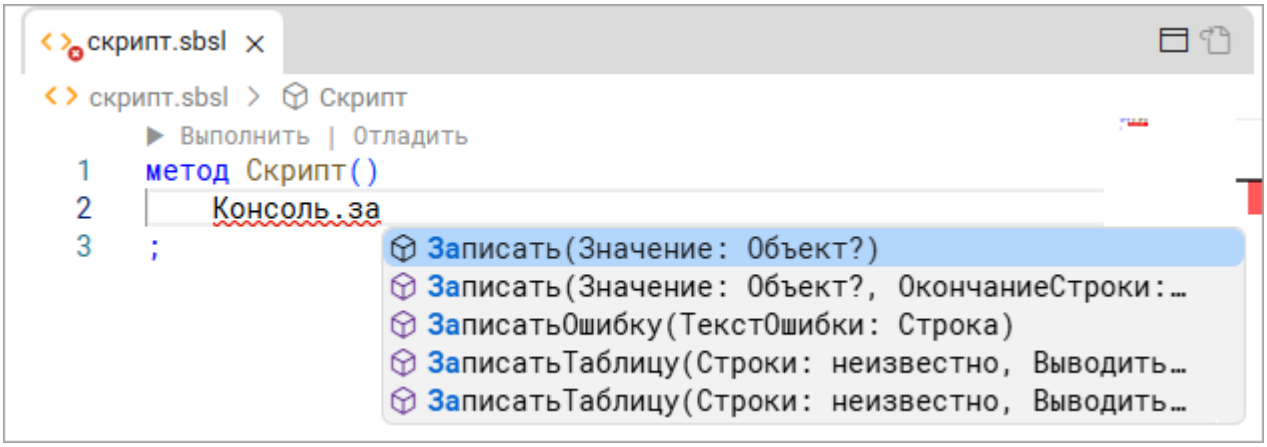
2. Нажмите **Главный метод**. Среда разработки «1С:Предприятие.Элемент» вставит шаблон метода **Скрипт()** и переместит курсор во вторую строку с отступом.



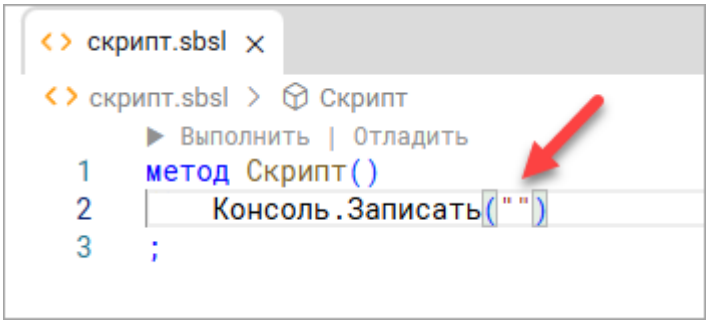
3. Начните писать **конс** — появится контекстная подсказка, нажмите в ней **Консоль**.



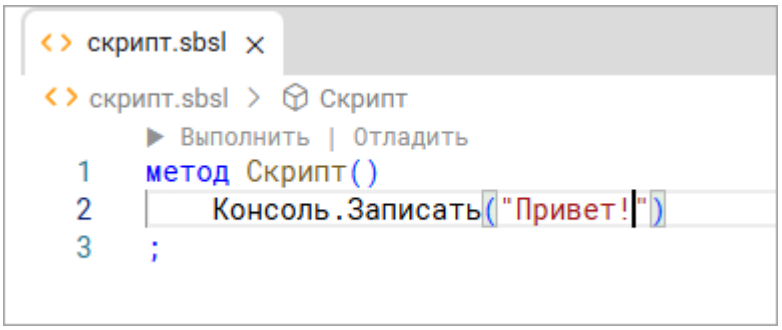
4. Поставьте точку и начните писать **за**. Появится контекстная подсказка, нажмите в ней **Записать(Значение: Объект?)**.



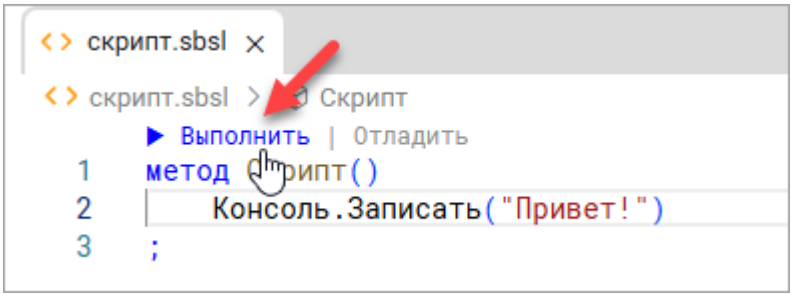
5. Нажмите двойную кавычку «"». Среда разработки автоматически удвоит её так, чтобы вы могли написать текст в кавычках.



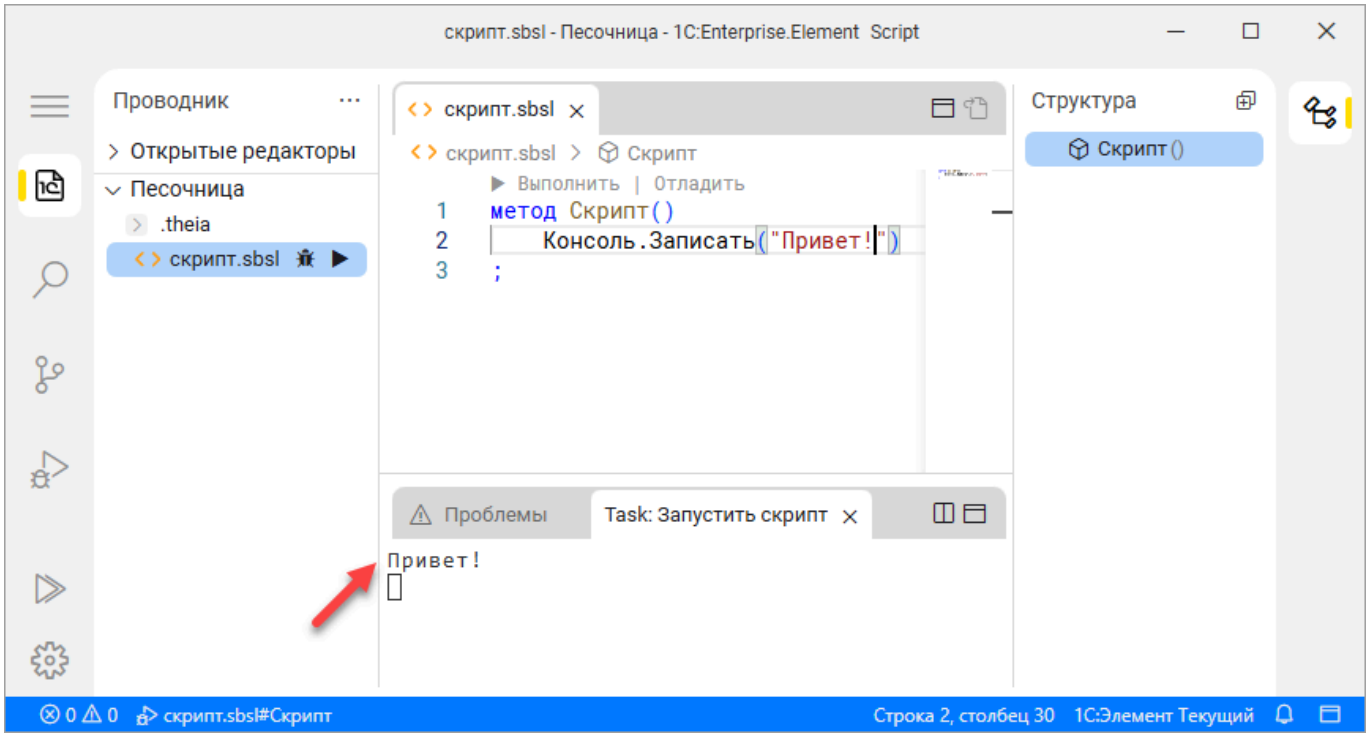
6. Напишите **Привет!**



7. Нажмите **Выполнить** над строкой **метод Скрипт()**.



8. В нижней части окна появится терминал, а в нём сообщение **Привет!**



Ваш первый скрипт работает!

Основные понятия

Файл **скрипт.sbsl**, который вы создали, называется *скриптом*, так же как и сама программа, которая в нём находится. Скрипт — это специализированная программа, написанная на высокоуровневом языке. От «обычных» программ она отличается своим назначением. Обычно скрипт выполняет набор рутинных или служебных действий, связанных с эксплуатацией, взаимодействием, обменом данными, настройками, управлением и обслуживанием информационных систем. Скрипт «1С:Элемента» — это текстовый файл, который всегда имеет расширение **.sbsl**.

Язык программирования, на котором вы написали свой скрипт, называется «*1С:Элементом*». «1С:Элемент» — это объектно-ориентированный язык, придуманный фирмой «1С». Он имеет статическую типизацию, стандартную библиотеку и дополнительные возможности из функционального программирования. Сейчас для вас это просто незнакомые слова, но далее вы познакомитесь с большинством его возможностей.

С одной стороны, «1С:Элемент» имеет много общих черт с другими языками программирования. С другой стороны, «1С:Элемент» специально разрабатывался так, чтобы его могли использовать не только профессиональные программисты. Например, все ключевые слова и инструкции языка имеют русское написание. Вам не нужно знать английский язык, для того чтобы написать собственную программу.

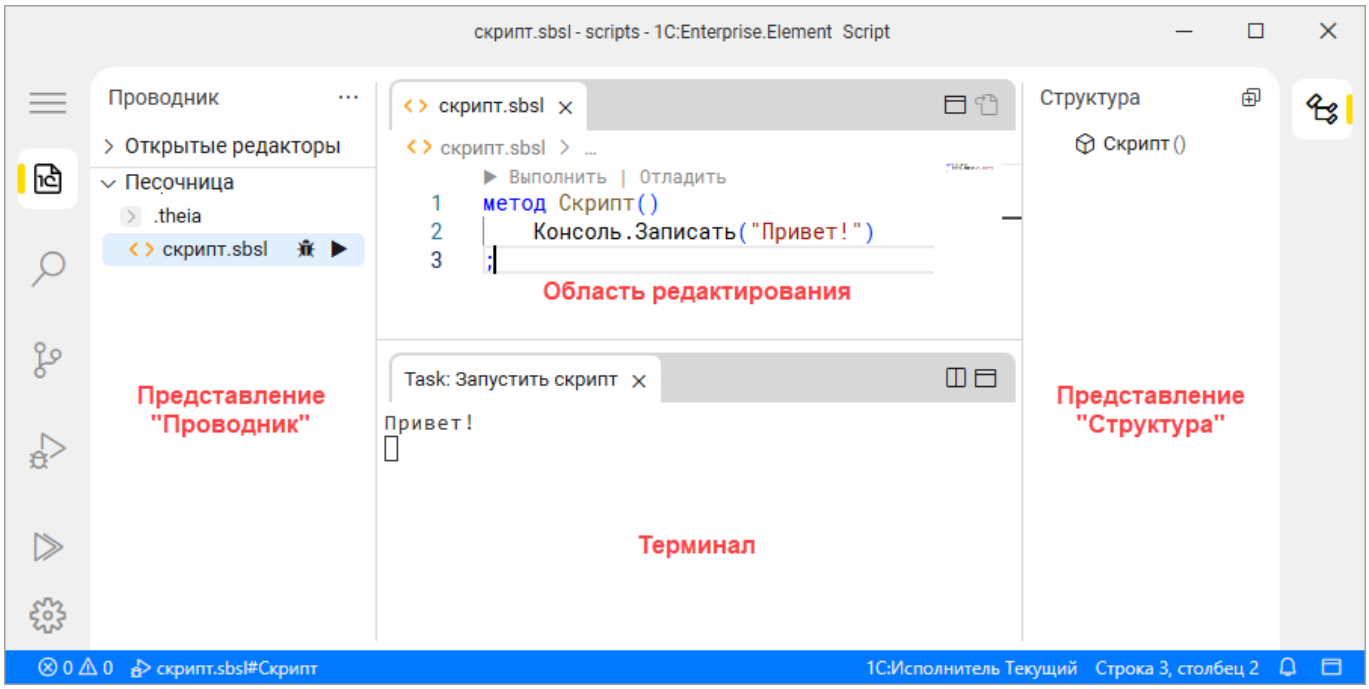
Программа, с помощью которой вы написали текст своего скрипта и запустили его на выполнение, — это среда разработки. Она называется «*1С:Предприятие.Элемент*». Это современная среда разработки, которая позволяет создавать, модифицировать и отлаживать скрипты. Она имеет развитые средства синтаксической помощи и контроля. Эта среда может функционировать не только локально, на вашем компьютере, но и в Интернете. Таким образом, вы можете разрабатывать скрипты удалённо, через Интернет, на неподготовленном компьютере. Достаточно иметь только браузер.

Движок, который исполнил код, написанный в вашем скрипте, называется «*1С:Предприятие.Элемент Скрипт*». Он является частью облачной технологии low-code разработки веб-кабинетов, порталов, браузерных и мобильных приложений. В этой книге вы не будете касаться вопросов разработки приложений, имеющих пользовательский интерфейс. Вы сосредоточитесь только на изучении языка программирования «1С:Элемент».

Глава 3. Настройка рабочего пространства

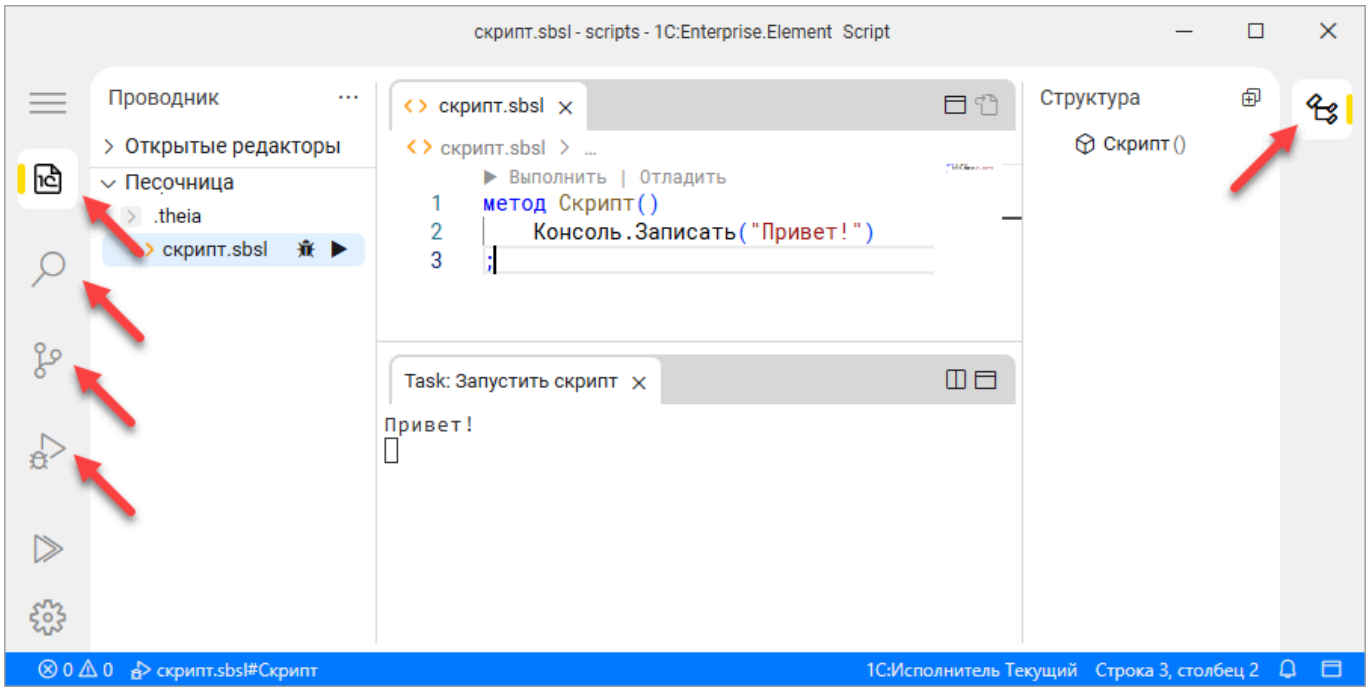
Обзор интерфейса среды разработки

Интерфейс среды разработки «1С:Предприятие.Элемент» состоит из *области редактирования* и *представлений*.

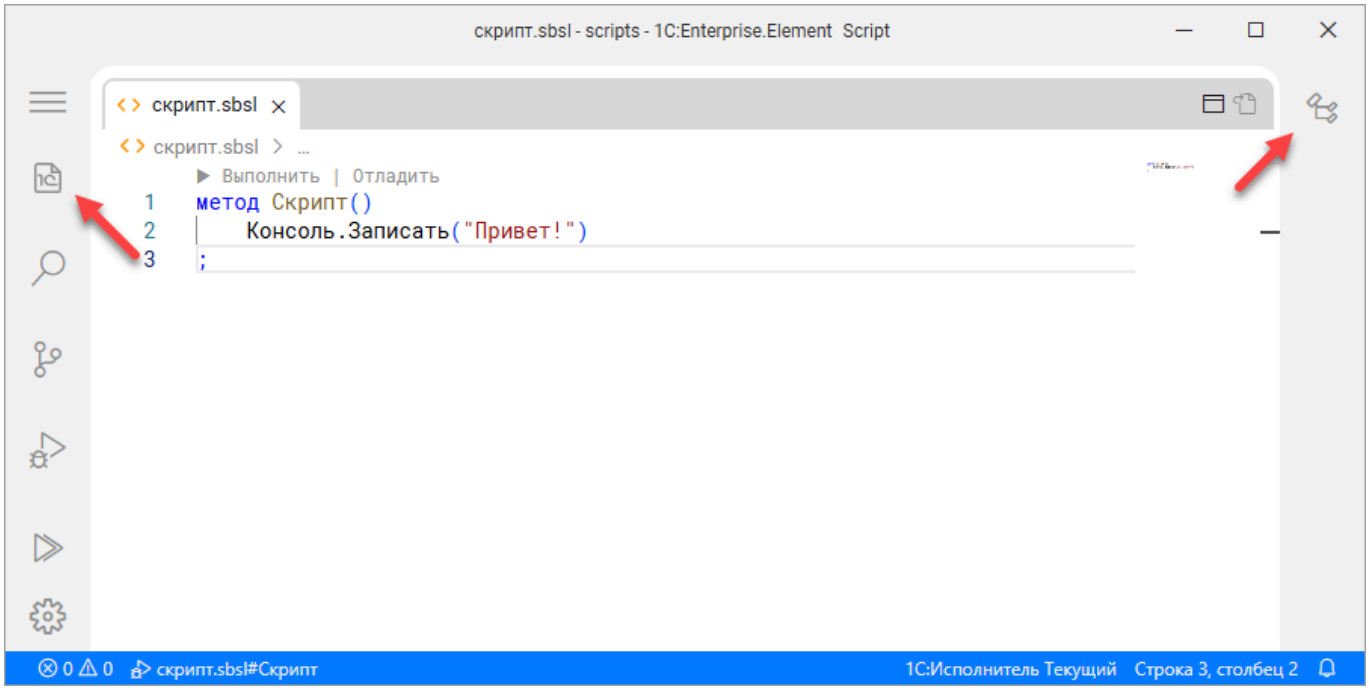


Представления — это «рабочие инструменты», предназначенные для выполнения конкретной задачи: для работы с файлами проекта, для отладки, для анализа структуры скрипта и т. д. Представления могут открываться слева, справа или внизу.

Есть представления, которые используются наиболее часто. Такие представления можно открыть быстро, для них слева и справа, на панелях действий, есть значки.

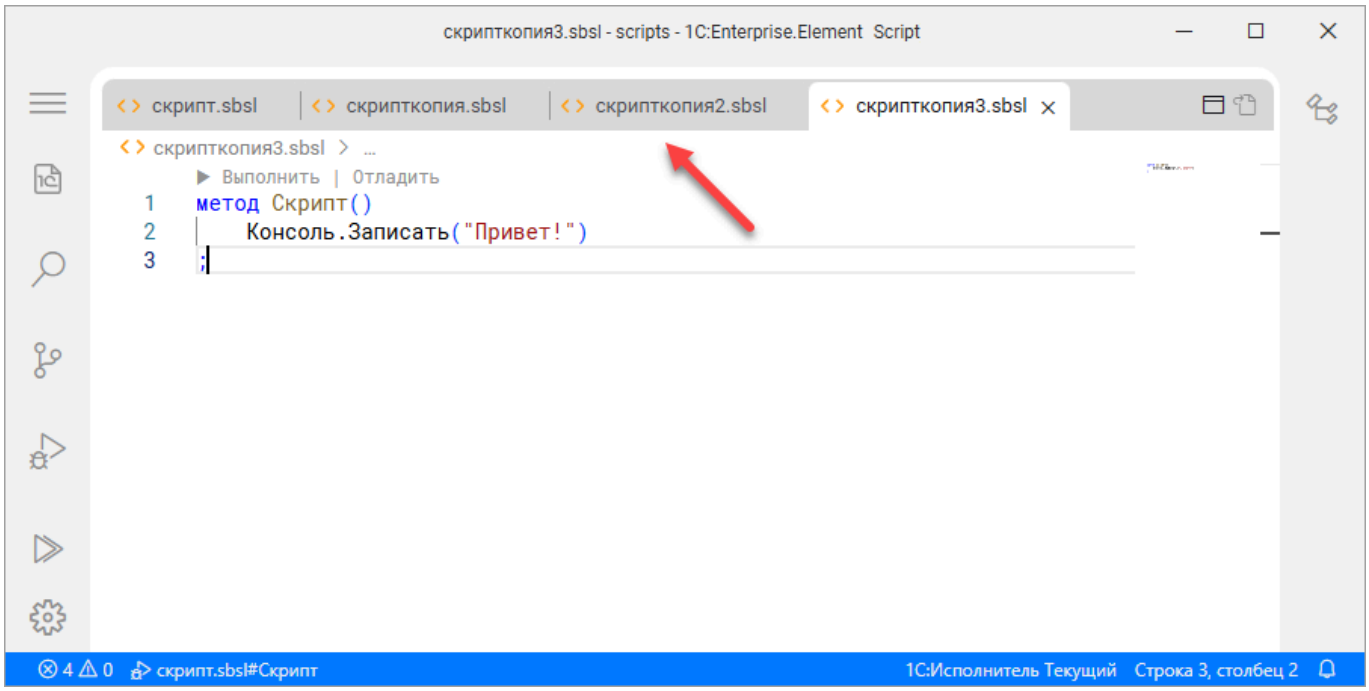


Эти значки работают как выключатели — повторное нажатие скрывает открытое представление. Это позволяет вам оперативно увеличивать ширину редактора, расположенного в центре.

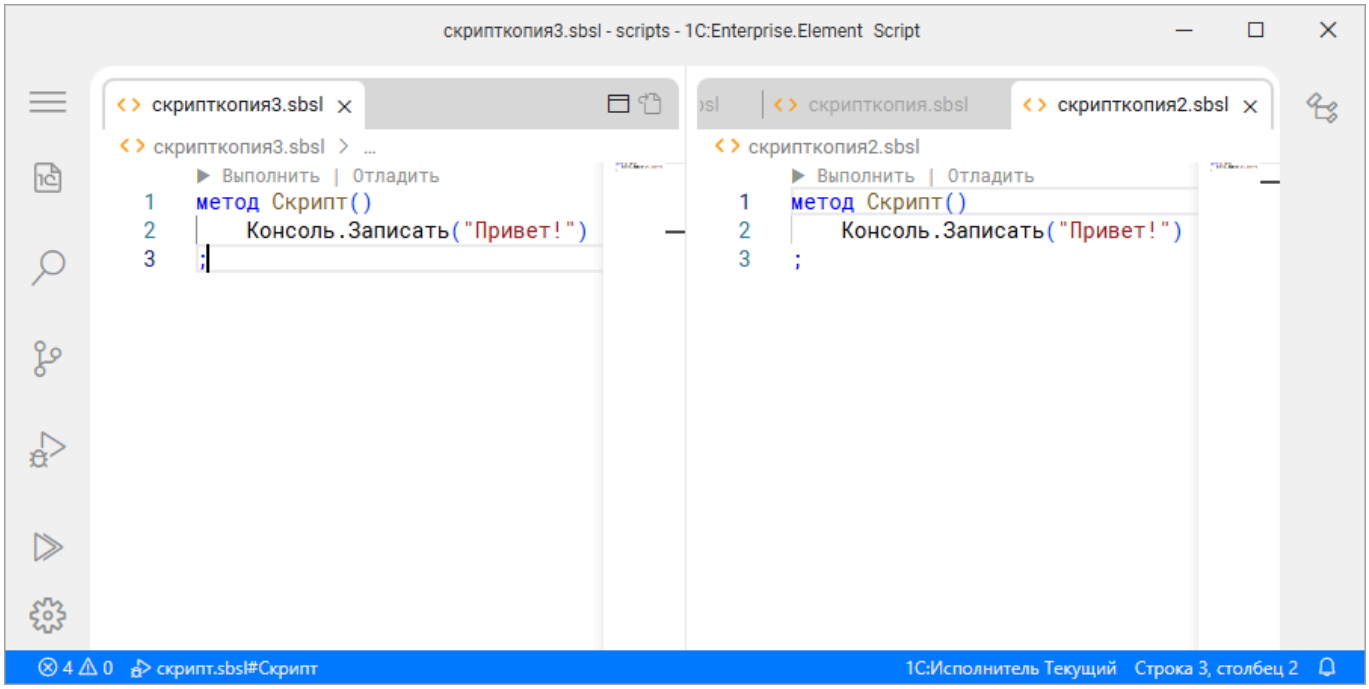


Остальные панели можно открыть из главного меню **Главное меню > Вид > Открыть представление**.

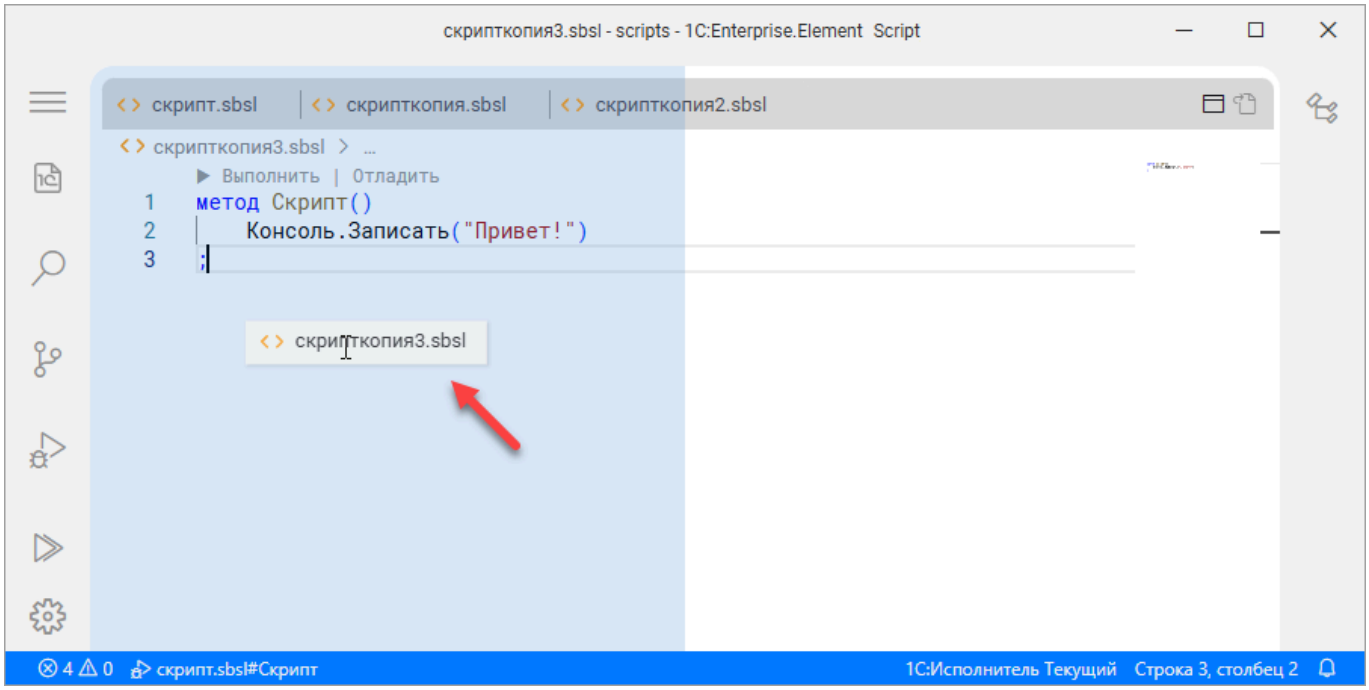
Область редакторов может содержать несколько редакторов, то есть можно редактировать несколько файлов одновременно.



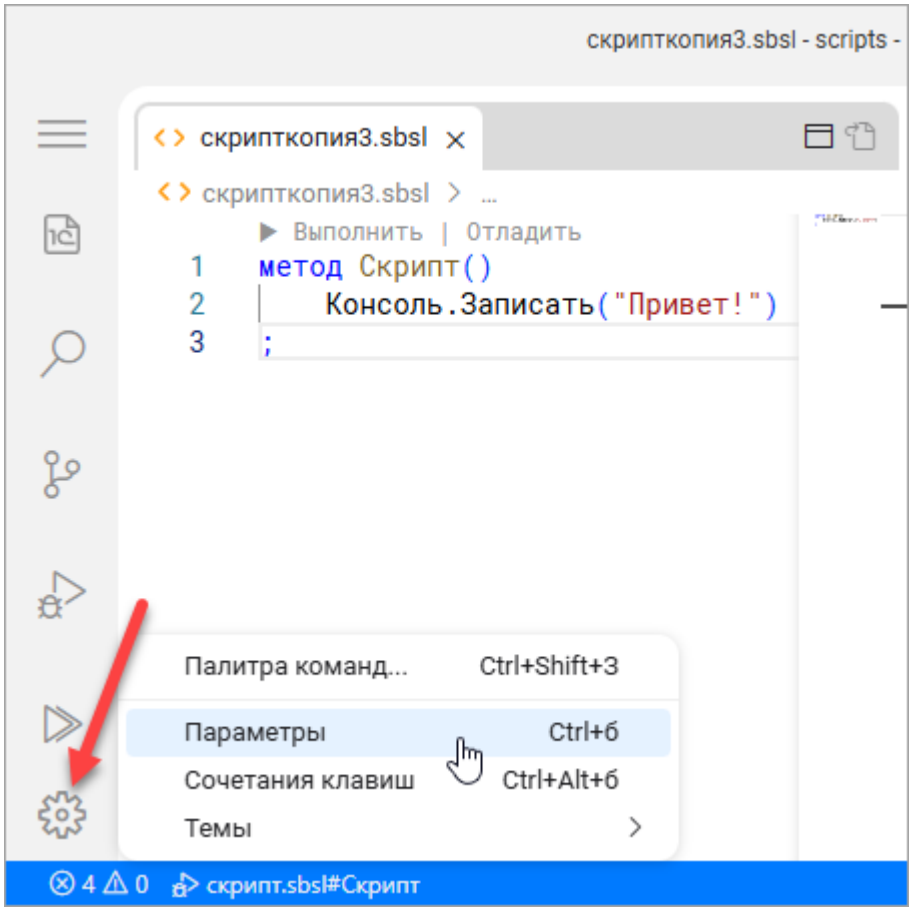
Вы можете самостоятельно располагать окна редакторов одно поверх другого или рядом друг с другом, вертикально или горизонтально.



Для этого потяните вкладку редактора в нужное вам место.

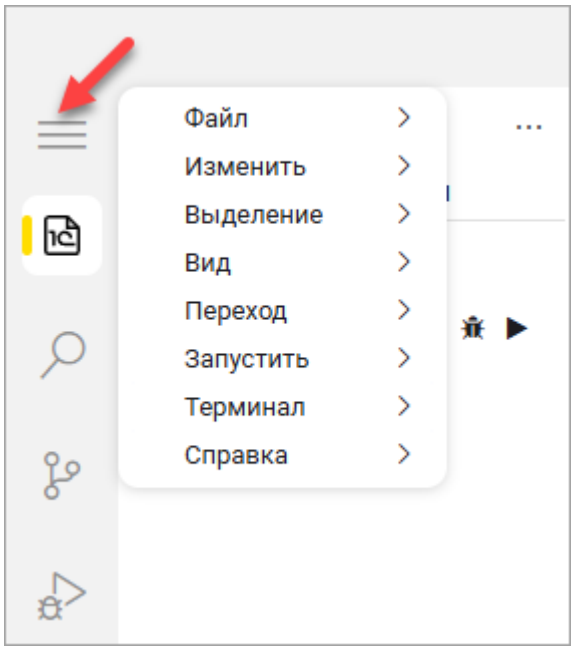


Слева в нижней части панели действий находится «шестерёнка» **Управление**, которая позволяет вам открыть **Параметры** — настройки среды разработки. Вы можете настроить внешний вид среды, свойства редакторов и другие параметры.



В нижней части среды разработки находится строка состояния. В левой её части отображается информация о выполняемых действиях, ошибках и предупреждениях, а в правой части — строка и столбец, в которых находится курсор в редакторе.

В любом случае, если вы забыли, где что включается, или если вам чего-то не хватает, все возможности среды разработки доступны из главного меню.



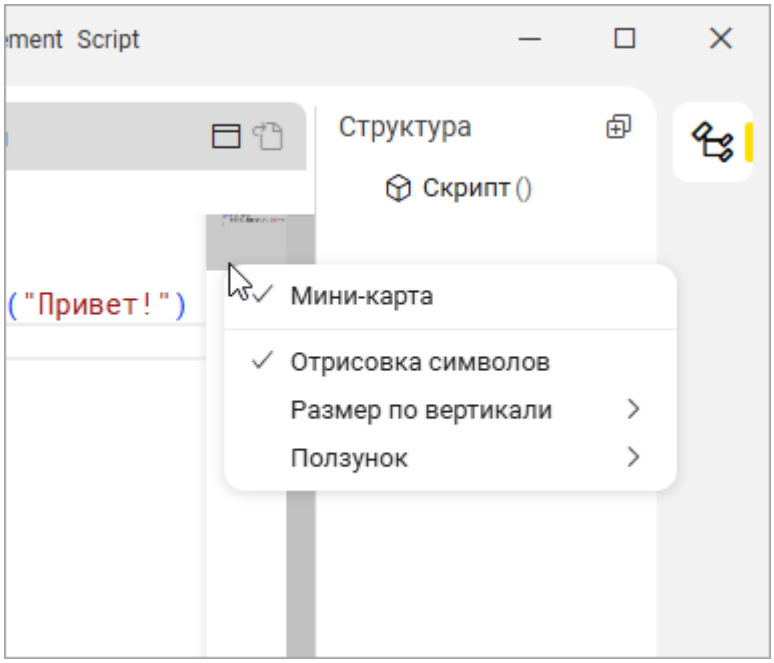
Настройка рабочего пространства

Прежде чем перейти к изучению языка и к программированию, настройте своё рабочее пространство, чтобы чувствовать себя удобно.

Отключите мини-карту. Мини-карта — это вертикальная полоса на правом краю области редакторов. Она полезна при работе с длинными скриптами, потому что показывает ваше положение внутри длинного файла и позволяет быстро перемещаться вверх и вниз по файлу.

Ваши примеры не будут большими, поэтому можете смело скрыть мини-карту, чтобы получить дополнительное пространство для редактирования.

Чтобы сделать это, вызовите контекстное меню на мини-карте и снимите флажок.

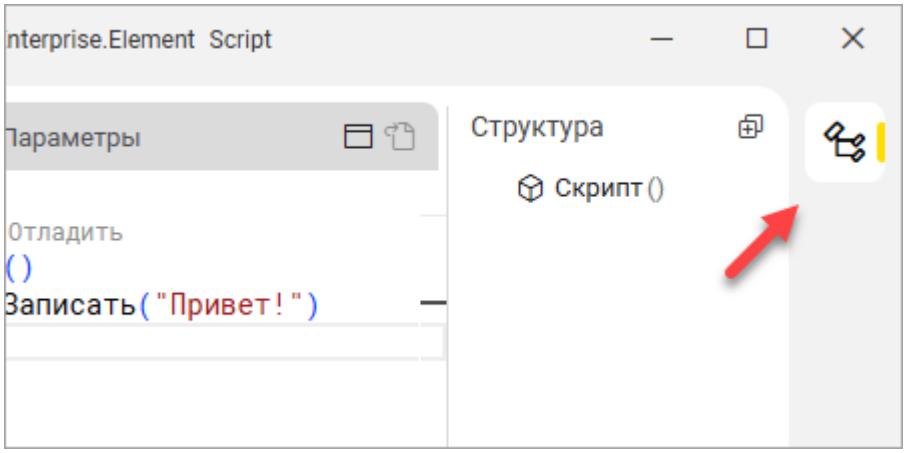


Те же действия можно выполнить через главное меню: **Главное меню > Вид > Переключить мини-карту**.

Скройте представление «Структура». Представление **Структура** открыто в правой вертикальной панели. Оно показывает те методы, которые есть в вашем скрипте. Это удобно при работе с большими скриптами, так как позволяет быстро перемещаться между разными методами.

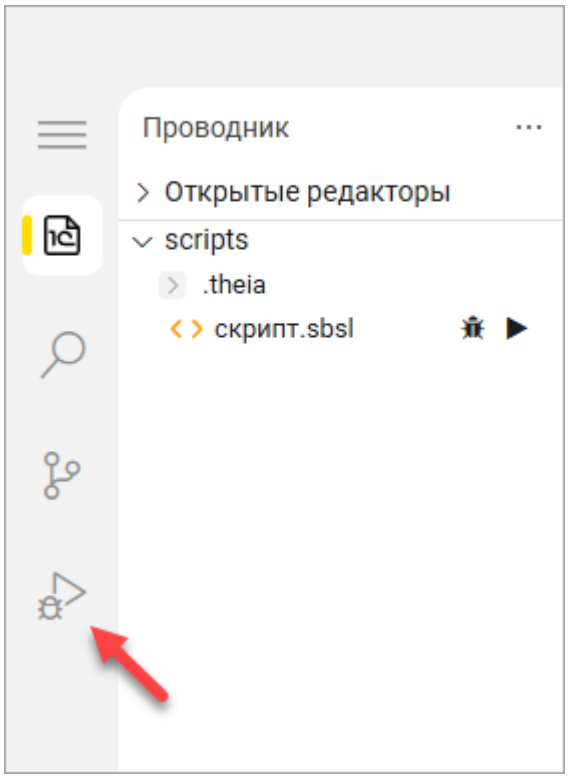
В ваших скриптах будет максимум два метода, поэтому можете смело скрыть представление **Структура**, чтобы получить дополнительное пространство для редактирования.

Чтобы сделать это, нажмите на значок этого представления в правой панели действий.

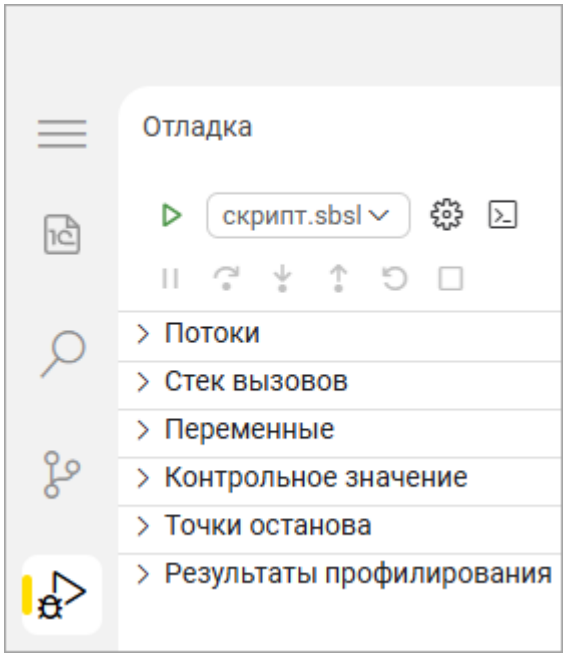


Те же действия можно выполнить через главное меню: **Главное меню > Вид > Структура**.

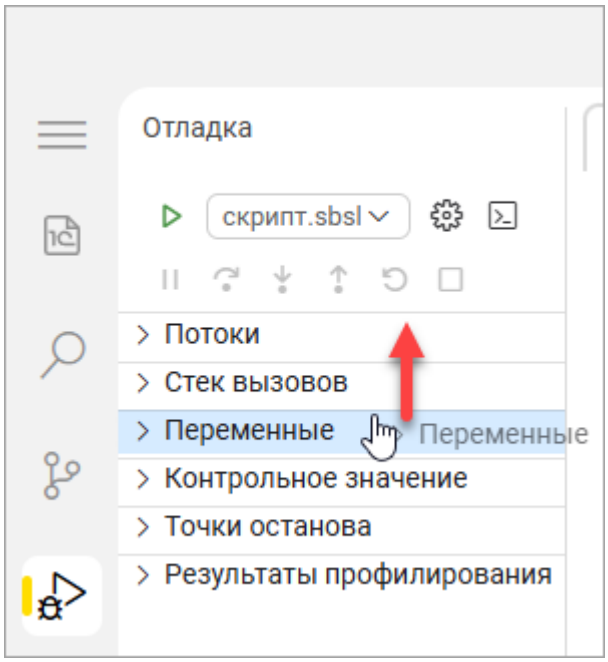
Измените порядок разделов в представлении «Отладка». Отладка — это специальный режим исполнения скрипта, в котором вы можете просматривать значения переменных, чтобы найти и устранить ошибки в работе скрипта. Обычно среда разработки «1С:Предприятие.Элемент» автоматически открывает это представление, когда происходит останов исполнения скрипта. Но вы можете открыть его вручную, нажав на значок в левой панели действий.



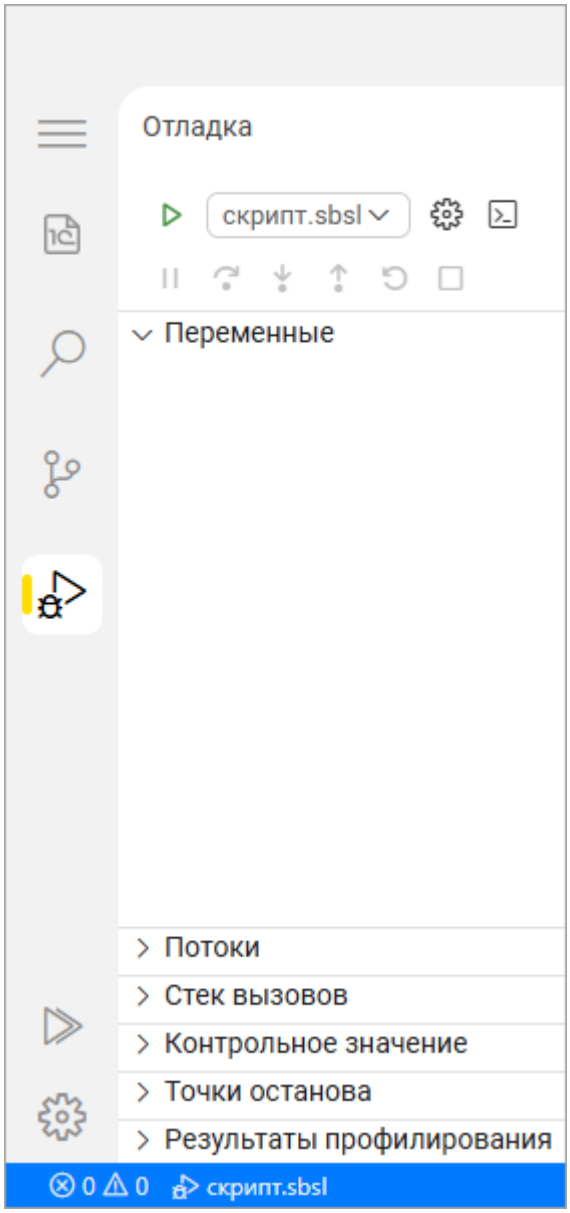
Представление **Отладка** содержит несколько разделов, и сейчас все они закрыты. Вам для отладки прежде всего будут нужны переменные. Поэтому ваша задача — переместить раздел **Переменные** на самый верх и раскрыть его.



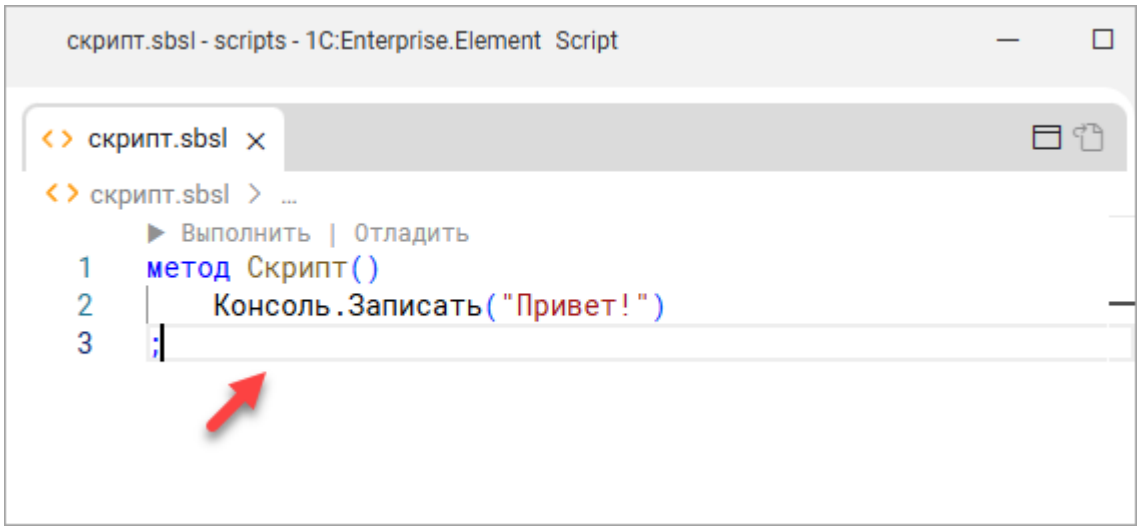
Схватите мышью заголовок раздела, перетащите его вверх.



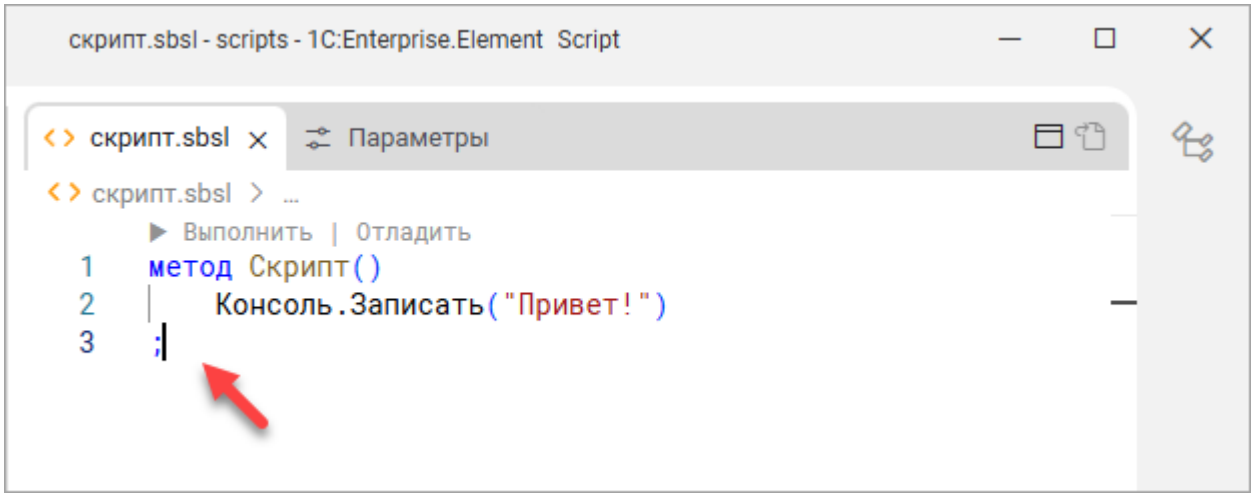
Нажмите на стрелку в начале заголовка — раздел раскроется, а остальные разделы опустятся вниз.



Отключите выделение текущей строки в редакторе. Это не влияет на удобство выполнения примеров, но, если вам не нравится выделение текущей строки в редакторе, вы можете отключить его.

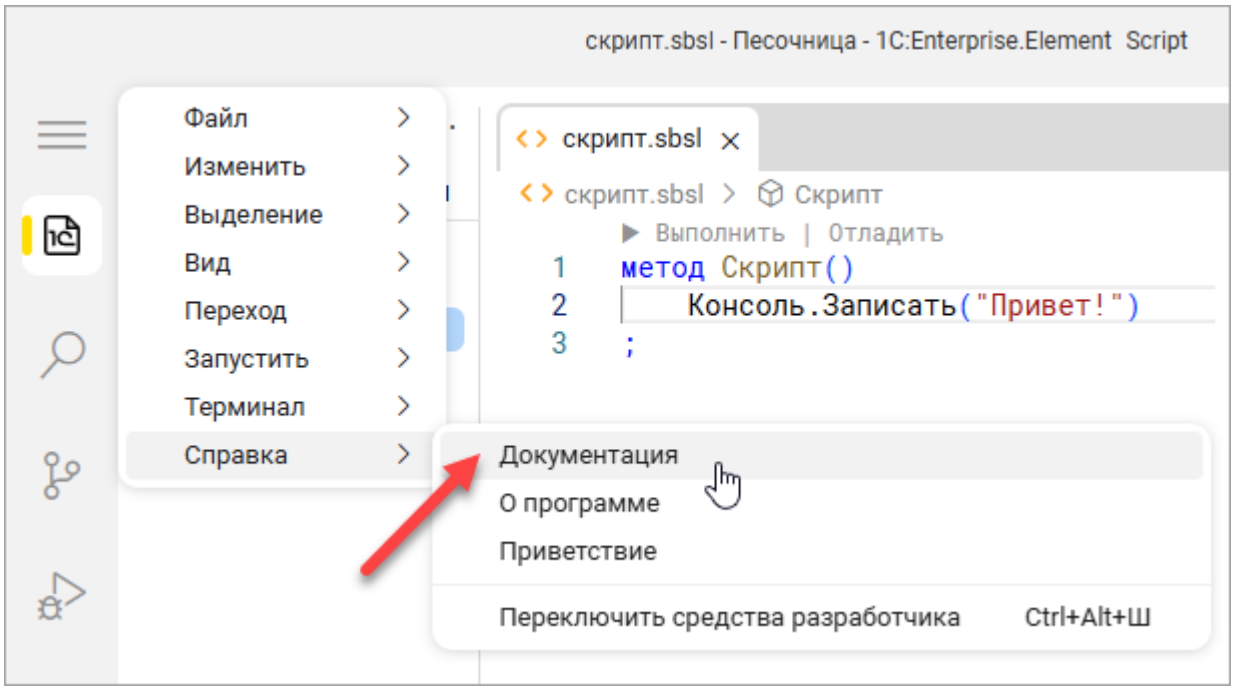


Для этого нажмите «шестерёнку» **Управление** слева внизу: **Управление > Параметры > Редактор > Режим подсветки выделения строки > Выключено**. После этого вы будете видеть только курсор без подсветки строки.



Справочная информация

По мере изучения языка «1С:Элемент» вам понадобится обращаться к справочным данным по языку и по типам, которые он использует. В книге будут ссылки на нужные интернет-страницы. Помимо этого вы самостоятельно можете открыть руководство разработчика — **Главное меню > Справка > Документация**.



Руководство разработчика — это часть документации. Оно рассказывает об устройстве языка «1С:Элемента», о том, как пользоваться его инструкциями и операциями. Также оно рассказывает о том, как использовать типы, содержащиеся в стандартной библиотеке, поставляемой в составе движка «1С:Предприятие.Элемент Скрипт».

Глава 4. Базовый уровень

Базовые понятия

Значение

Что делает программа? Вы можете сказать, что любая программа содержит последовательность команд, которые выполняются друг за другом. Да, это правильно, если смотреть на программу издалека.

Если взглянуть на неё более внимательно, вы увидите, что программа выполняет действия со *значениями*. Все эти действия делятся на три большие группы.

Сначала программа **получает** какие-то значения. Может быть, эти значения были записаны на жёстком диске — тогда она читает их с диска. Может быть, вы сами ввели эти значения с клавиатуры. Есть много способов, которыми программа может получить значения. Это не важно, главное, что программа их откуда-то получает.

Затем программа **что-то делает** с теми значениями, которые она получила. Например, вы ввели время начала занятий и время окончания занятий. Тогда программа может из одного значения вычесть другое и получить третье значение — продолжительность ваших занятий. Другой пример — вы ввели значение 101. Это номер кабинета, в котором будут проходить занятия. Программа может проверить, есть у вас в списке такой кабинет или ещё нет.

И, наконец, любая программа завершает свою работу тем, что **выводит** значения. То есть показывает их вам на экране. Записывает их на жёсткий диск. Печатает на принтере. Передаёт другому компьютеру. Конкретный способ тоже не важен, важно лишь то, что значения являются ещё и результатом деятельности программы.

Тип

В компьютере может существовать много разных значений. Чтобы программа могла что-то делать с этими значениями, она должна знать их *тип*.

С типами вы сталкиваетесь в обычной жизни. Например, чтобы записывать номера кабинетов, в которых проходят занятия, вы используете тип `Число`, потому что номер кабинета — это набор цифр. Чтобы записывать фамилии учителей, вы используете тип `Строка`, потому что фамилия учителя — это последовательность букв.

Когда вы вводите в компьютер номер кабинета — для него это одно значение. Когда вы вводите фамилию учителя — это для него другое значение. Программа нужна для того, чтобы что-нибудь сделать с этими значениями. Как вы думаете, что будет, если программа попытается сложить 101 и «Казаков К. Д.»? Или попытается разделить «Герасимова Е. С.» на «Давыдова А. В.»? Ничего хорошего из этого не получится.

Чтобы вышел какой-то толк, программа прежде всего должна знать, какой тип имеют те значения, с которыми ей предстоит работать.

Например, если это числа, то их можно складывать, делить, умножать, вычитать. Если это строки, то умножать и делить их нельзя. Их можно только сложить. На языке компьютера это будет означать, что к одной строке нужно дописать другую строку.

А вот если значения имеют разные типы (например, одно значение имеет тип `Число`, а другое — тип `Строка`), то с такими значениями вообще никаких действий произвести нельзя.

Для чего ещё программе нужно знать тип значений? Чтобы записать их на диск, например. Потому что числа хранятся одним образом, а строки — другим. А если вы захотите записать на диск картинку, то программа выберет для этого третий способ хранения данных.

> **Важно:**
>
> Каждое значение относится к какому-либо типу. Говорят: «Значение имеет тип `Строка`» или «Значение имеет тип `Число`».

Представление

Пока программа записывает значения на диск или передаёт их другому компьютеру, она это делает так, как удобно ей, и в такой форме, которая ей удобна. Когда эти значения нужно показать на экране или напечатать, программа не может делать это так, как ей хочется. Она должна сделать это так, чтобы любому человеку было понятно, что это за значения.

Как раз для этого и нужно *представление*. Каждый тип имеет собственное представление. Представление — это способ, которым значения этого типа нужно показывать на экране.

Например, представление типа `Число` — это последовательность цифр. В этой последовательности может встретиться одна запятая. Она отделяет целую часть числа от дробной части. Например, 346,45 или 58,2.

Также в этой последовательности может встретиться пробел. Он отделяет друг от друга группы разрядов. Это помогает легче читать большие числа. Например, 1 475 или 395 987,41.

У типа `Строка` представление другое. Это просто последовательность любых символов. В этой последовательности тоже может встретиться пробел или запятая. Но никакого особенного смысла они не имеют. Например, «Давыдова А. В.» или «математика, мой любимый предмет».

> **Важно:**
>
> Любой тип имеет представление. Это правило, по которому обозначаются значения этого типа. Любое значение тоже имеет своё представление. Это набор символов, которым обозначается это значение на экране или на печати.

Почему текст разноцветный

Вернитесь к скрипту, который вы написали в самом начале при первом запуске среды разработки «1С:Предприятие.Элемент».

Почему слова, которые вы написали в скрипте, разного цвета? Почему некоторые из них синие, некоторые чёрные, а некоторые даже бордовые?

```
метод Скрипт()
    Консоль.Записать("Привет!")
;
```

Всё просто: они разного цвета для того, чтобы вы не запутались. Для того, чтобы вам было легче читать то, что написано. Дальше вы увидите, что слова могут быть даже зелёные и фиолетовые.

Когда вы что-то пишете в скрипте, тем самым вы объясняете компьютеру, что он должен сделать. Объясняете на языке «1С:Элемента», который понятен и ему, и вам. Этот язык похож на наш обычный язык, но он гораздо проще. В нём есть только такие предложения, которые выражают приказ, просьбу или совет.

Эти «предложения», которые вы пишете в скрипте, называются *инструкциями*. Порядок инструкций важен, потому что компьютер «читает» их одну за другой, в той последовательности, в которой они написаны.

В обычной жизни вы отделяете одно предложение от другого точкой и пробелом. В «1С:Элементе» никакие специальные символы в конце инструкции не ставятся, но каждая инструкция должна быть написана с новой строки.

Сейчас вы написали одну инструкцию **Консоль.Записать("Привет!")**. Чёрным цветом среда разработки «1С:Предприятие.Элемент» раскрасила **Консоль** и **Записать** — это нечто, что среде разработки известно. Нечто, что она знает, умеет и может делать без дополнительных объяснений. Вспомните: она подсказывала вам эти слова в своей контекстной подсказке.

Что среда разработки раскрашивает синим цветом? Синим цветом она раскрашивает те символы и слова, которые являются обязательными, без которых не получится правильная инструкция.

Например, синим цветом она раскрашивает круглые скобки, которые находятся в конце строки. Среда разработки знает, что такое **Консоль.Записать**. Она знает, что после этого обязательно должна быть указана строка, которую нужно записать. Эту строку она будет искать внутри скобок, которые обязательно должны быть в такой инструкции.

Ещё синим цветом она раскрасила слово **метод** в начале и точку с запятой в конце, потому что это начало и окончание составной инструкции.

Теперь осталось разобраться с бордовым цветом. С ним всё просто. Бордовым цветом среда разработки выделяет значение. Значение, которое вы написали прямо в тексте программы. Такие значения, написанные прямо в тексте программы, называются *литералами*.

В вашем примере написан литерал строкового значения, он обрамляется двойными кавычками «"», а внутри содержит последовательность символов.

> **Примечание:**
>
> В «1С:Элементе» в простых инструкциях никакие специальные символы в конце инструкции

не ставятся. Однако есть составные инструкции и блоки инструкций, которые заканчиваются обязательным символом точка с запятой «;». Об этом вы узнаете позже (81).

В вашем скрипте есть составная инструкция — `метод`.

```
метод  Скрипт()

;
```

Инструкции

В русском языке вы можете составлять разные предложения. Всё зависит от вашего желания, словарного запаса и привычек.

В «1С:Элементе» всё проще. Нельзя писать какие угодно инструкции. Существует небольшое количество инструкций, которые можно разбить на 8 групп. Это:

- Объявление переменной.
- Присваивание.
- Условные инструкции: `если`, `выбор`.
- Инструкции цикла: `для по`, `для из`, `пока`.
- Описание метода.
- Вызов метода.
- Исключения: `исключение`, `попытка`, `выбросить`.
- Инструкция выделения блока кода.

Инструкция, которую вы написали в своём скрипте, **Консоль.Записать("Привет!")**, это инструкция вызова метода.

Инструкция объявления переменной

Переменная — это специальное место в памяти компьютера. В этом месте можно сохранить одно значение на то время, пока выполняется ваша программа.

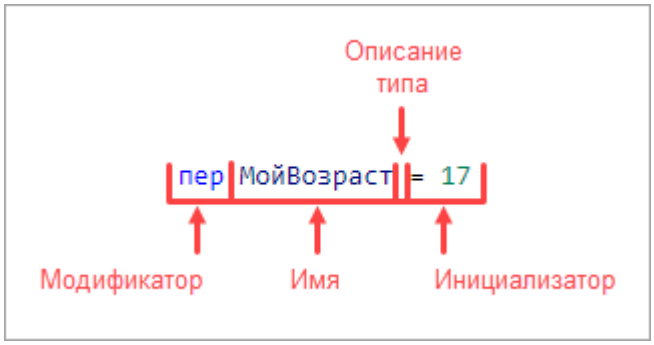
Чтобы создать переменную, нужно написать инструкцию объявления переменной. Например:

```
пер  МойВозраст = 17
```

Напишите её в своём скрипте вместо **Консоль.Записать("Привет!")**. У вас получится следующий скрипт.

```
метод  Скрипт()
    пер  МойВозраст = 17
;
```

Инструкция объявления переменной может выглядеть по-разному, сложнее, чем в этом примере. В общем случае она может состоять из модификатора, имени переменной, описания типа (здесь его нет) и инициализатора.



Совет:

Подробнее об инструкциях читайте в руководстве разработчика. Слева в рубрикаторе — раздел **Синтаксис - Инструкции**.

Модификатор

Модификатор — это обязательное ключевое слово, которое указывается в начале объявления переменной. Используются три модификатора:

- `пер` (от английского variable — переменная) — переменная, доступная для записи и для чтения. После создания такой переменной можно изменить её значение. Например:

  ```
  пер МойВозраст = 17
  ```

  Это самый распространённый модификатор, он используется в подавляющем большинстве случаев.

- `знч` (от английского value — значение) — переменная, доступная только для чтения. После создания такой переменной изменить её значение нельзя, его можно только прочитать. Например:

  ```
  знч МойВозраст = 17
  ```

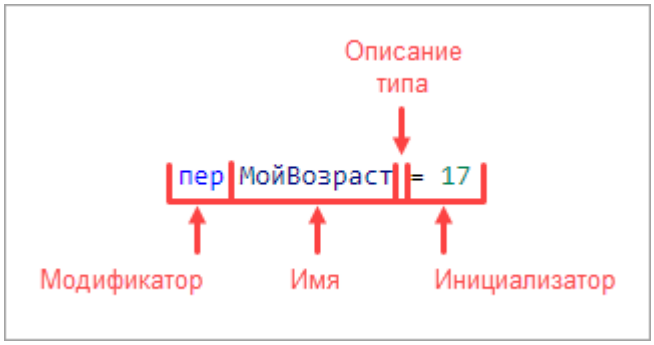
  Такой модификатор используется в тех случаях, когда вы хотите быть уверенными в том, что где-то в коде программы вы случайно не измените это значение.

- `исп` (от английского use — использование) — говоря простым языком, в таких переменных хранят сложные значения (экземпляры), которые удерживают системные ресурсы (порты, файлы и т. д.). Это может быть, например, соединение по SSH-протоколу, результат вызова SQL-процедуры, поток чтения данных. Важным моментом является то, что после окончания работы с такой переменной нужно обязательно освободить тот системный ресурс, которым пользовался этот экземпляр. Это нужно для того, чтобы другие программы тоже могли использовать данный ресурс.

  Так вот, модификатор `исп` гарантирует, что после окончания работы с такой переменной движок «1С:Предприятие.Элемент Скрипт» освободит системный ресурс, даже если вы забудете это сделать.

  Говоря сложным языком, модификатор `исп` используют для переменных, тип которых является потомком типа `Закрываемое`. Такая переменная доступна только для чтения. При выходе из области видимости для такой переменной будет автоматически вызван метод `Закрыть()`. Но об иерархии типов *(103)*, областях видимости *(94)* и методах *(101)* вы узнаете позже.

Имя переменной



После модификатора пишется *имя переменной*, это тоже обязательный элемент инструкции объявления переменной. Имя нужно для того, чтобы отличить одну переменную от другой, а также чтобы прочитать значение переменной или записать в неё другое значение.

Имя должно начинаться с буквы и не должно содержать пробелов. Если хочется составить имя из нескольких слов, каждое следующее слово принято писать с большой буквы. Буква «ё» обычно в именах не используется, вместо неё надо писать букву «е».

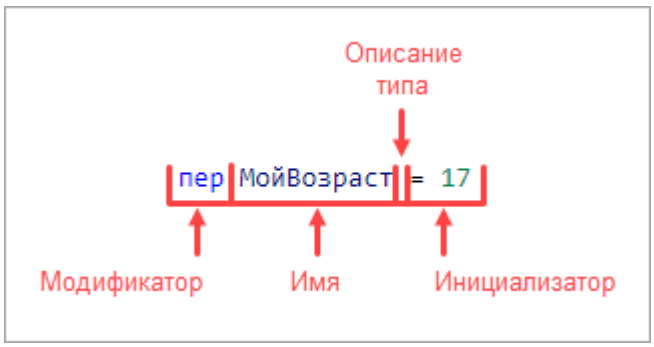
Придумывайте такие имена, чтобы они были понятны и вам, и другому человеку. Не стесняйтесь использовать несколько слов для составления имени. Два, три слова — это нормально.

Когда имя очень короткое или оно является каким-то сокращением, другому человеку будет трудно понять, что вы имели в виду. Да и вы, если посмотрите на свою программу через несколько месяцев, с трудом вспомните, что вы хотели сказать таким именем. Вот, например, плохие имена: **доб**, **стро**, **квоТТ**.

Очень длинное имя — это тоже нехорошо. Например, **ДеньгиНалИБезНалМеждуНамиИКонтрагентами**. Использовать такую переменную в каких-нибудь формулах будет очень неудобно.

В общем, старайтесь быть краткими и в то же время понятными.

Описание типа



Описание типа — это обозначение, которое говорит, что значение, находящееся в переменной, принадлежит к такому-то типу. Описание типа может быть, а может и не быть. В вашем случае, как видите, описание типа отсутствует, но это не значит, что в каких-то ситуациях можно не знать тип переменной.

```
пер МойВозраст = 17
```

В «1С:Элементе» используется *статическая типизация*. Это значит, что при объявлении переменной, параметра *(83)* или метода *(81)*, возвращающего значение, сразу должен быть указан тип. В результате ещё в момент компиляции (до того, как скрипт начнёт выполняться) проверяется соответствие типов переменных, параметров и т. д. и присваиваемых им значений. То есть вы не можете в переменную, описанную с типом `Число`, случайно записать значение типа `Строка`.

Таким образом, статическая типизация «1С:Элемента» позволяет увидеть и исправить многие ошибки ещё на стадии компиляции скриптов, что является несомненным преимуществом «1С:Элемента».

Почему же тогда в вашем примере отсутствует описание типа? Просто для краткости, для удобства. Вы не просто создаёте переменную, а сразу же указываете её значение, **17**, в виде литерала. В этом случае из текста программы видно, какой тип имеет значение, и описание типа можно не писать.

Но если бы вы захотели, то могли бы его и написать. В вашем случае оно выглядело бы так:

```
пер МойВозраст: Число = 17
```

Описание типа представляет собой двоеточие «:», пробел и имя типа. То есть описание типа это «: Число».

Имя типа — это обозначение, которое позволяет отличить один тип от другого. Все имена типов, которыми может оперировать «1С:Элемент», описаны в справке по стандартной библиотеке. Например, это типы Строка и Число.

Начальное значение

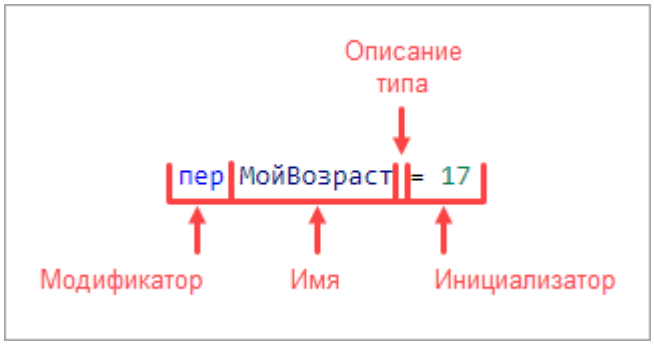
В некоторых случаях движок может сам вычислить значение переменной, которая объявляется. Дело в том, что некоторые типы имеют значения по умолчанию. Например:

- тип `Число` — 0;
- тип `Строка` — пустая строка (стока, не содержащая ни одного символа);
- тип `Булево` — `Ложь`;
- тип `Дата` — 1 января 0001 года;
- тип `Время` — 0 часов 0 минут 0 секунд 0 миллисекунд.

Поэтому если вы напишете так, как показано ниже, то в переменной **МойВозраст** будет значение **0**:

```
пер МойВозраст: Число
```

Инициализатор



Вы можете с помощью одной инструкции объявить переменную нужного вам типа. С помощью другой инструкции можете присвоить ей то значение, которое нужно.

В то же время «1С:Элемент» позволяет совместить два этих действия и сделать все в одной инструкции — для удобства и для краткости. Для этого в конце инструкции нужно написать инициализатор.

Инициализатор представляет собой знак равенства «=», пробел и выражение *(35)*. То есть в вашем случае инициализатор это **= 17**.

```
пер МойВозраст = 17
```

Что такое выражение — вы узнаете позже, сейчас это не так важно. Главное, что вам нужно понять: выражение — это нечто, что движок может вычислить, то есть получить значение выражения.

Инициализатор не является обязательным элементом инструкции объявления переменной, но его нужно указывать тогда, когда тип переменной не имеет значения по умолчанию. В этом случае среда разработки не может сама определить начальное значение переменной и необходимо явное его указание. Например, у типа `Версия` нет значения по умолчанию, поэтому в объявлении переменной типа `Версия` инициализатор необходим. Версия 8.0:

```
пер ВерсияТехнологии: Версия = Версия{8.0}
```

> **Примечание:**
>
> Фигурные скобки «{» и «}» можно вводить с помощью Alt-ввода *(227)*.

Ограничения

Не все слова из перечисленных ниже вам сейчас знакомы, но по мере изучения материала вы можете возвращаться к этим пунктам, чтобы освежить свои знания.

Объявление переменной может располагаться в любом месте метода *(101)*. При этом нужно учитывать следующие ограничения:

- Нельзя объявить сразу несколько переменных в одной инструкции.

```
пер МойВозраст: Число пер МойРост: Число
```

- Переменная, например, МойВозраст, не может быть объявлена дважды в одной области видимости *(94)*.

```
метод Скрипт()
    пер МойВозраст: Число
    пер МойВозраст: Число
;
```

- Имя переменной, например, Длина, не может совпадать с именем параметра метода *(83)*.

```
метод ВычислитьПлощадь(Длина: Число, Ширина: Число): Число
    пер Длина: Число
```

```
    возврат Длина * Ширина
;
```

• Переменная не может быть использована в собственном инициализаторе.

```
пер МойВозраст = МойВозраст + 5
```

• Не поддерживается объявление переменной без описания типа и инициализатора одновременно.

```
пер МойВозраст
```

> **Совет:**
>
> Подробнее об инструкции объявления переменной читайте в руководстве разработчика.

Теперь познакомьтесь с тем, как работает эта инструкция объявления переменной. Вы не будете никуда ничего выводить, чтобы где-то в программе можно было увидеть значение этой переменной. Вы сразу поступите, как настоящие профессиональные программисты. «Возьмёте в руки» отладчик и с его помощью заберётесь в самую глубь движка, чтобы увидеть, как все происходит на самом деле.

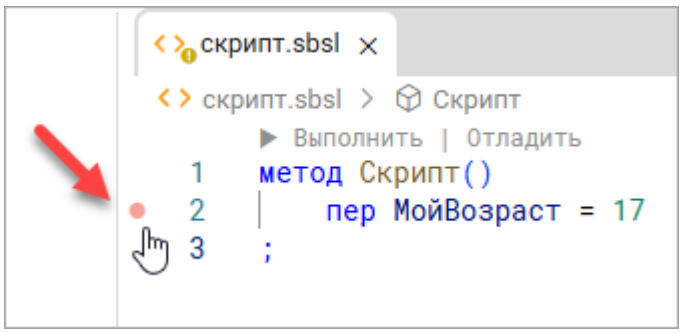
Это очень интересно!

Точки останова и просмотр значений

Никаких особенных действий, чтобы воспользоваться отладчиком, производить не нужно. Почти всё вам уже известно: как написать код, как запустить его на исполнение. Единственное, чего вы пока не знаете, это как сказать движку «1С:Предприятие.Элемент Скрипт», чтобы он остановился в нужном месте.

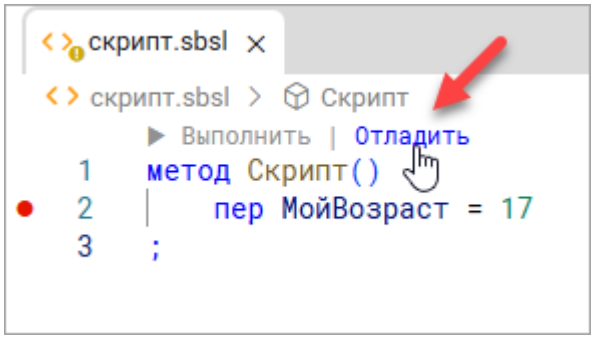
Для этого используются точки останова. *Точка останова* — это специальная отметка. Она показывает, перед выполнением какой инструкции движок должен остановиться и ждать дальнейших команд от вас.

Чтобы установить точку останова, нужно щёлкнуть мышью перед номером той строки, перед выполнением которой нужно остановиться.

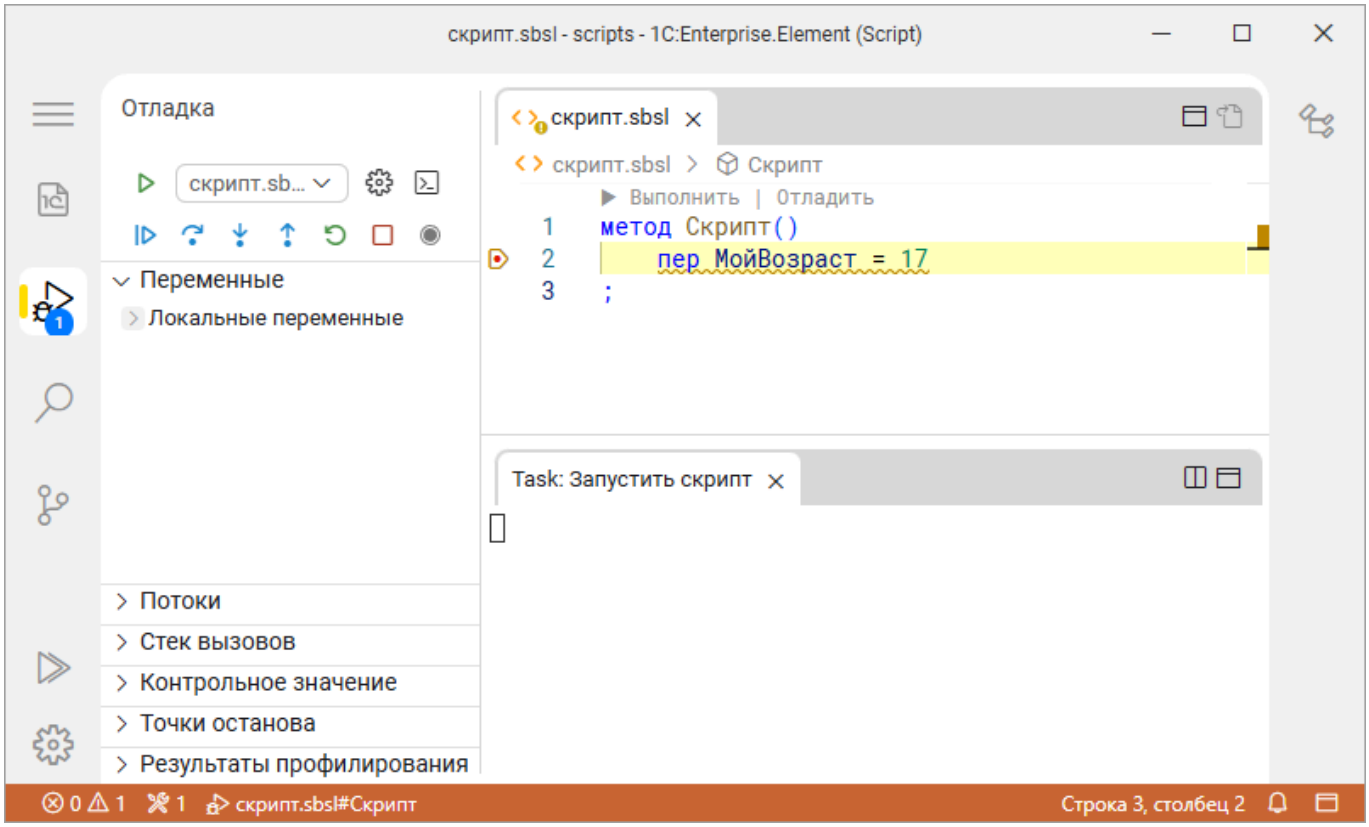


Если на точке останова снова щёлкнуть мышью, то точка останова пропадёт.

Теперь всё, что вам нужно, это запустить движок в режиме отладки. Нажмите **Отладить** над строкой **метод Скрипт()**.

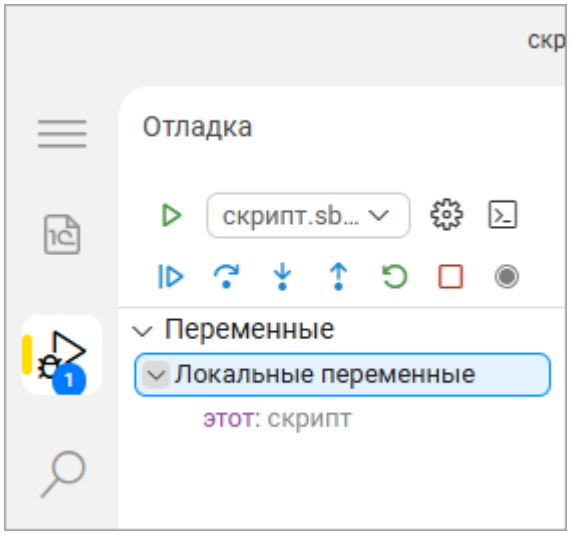


Как только движок дойдёт до выполнения инструкции, которую вы отметили, он остановится и переключится в представление **Отладка**. Вы увидите такую картинку.



Стрелка перед жёлтой строкой слева показывает, какую инструкцию движок приготовился выполнить. Сейчас он приготовился выполнить инструкцию присваивания, которую вы написали.

Как посмотреть значение переменной **МойВозраст**? Для этого есть несколько способов. Первый и самый простой — в представлении **Отладка** в разделе **Переменные** раскрыть группу **Локальные переменные**. Здесь отладчик автоматически покажет все переменные и их значения, которые доступны в данный момент.



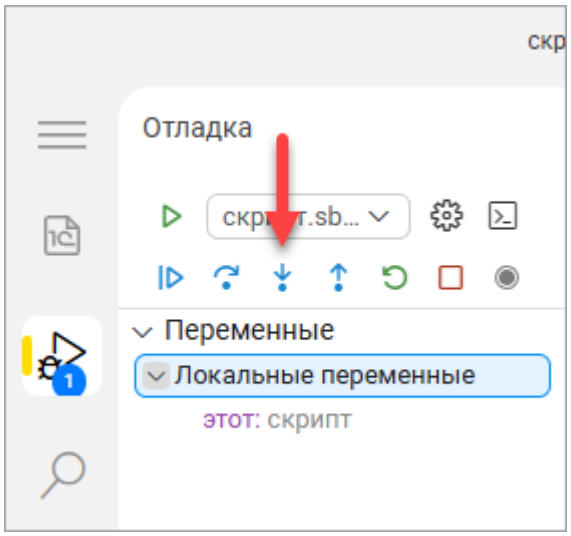
Как видите, здесь есть какая-то переменная **этот**, но нет переменной **МойВозраст**.

Переменная **этот** — это сам скрипт. Он тоже является программным объектом, и с ним можно выполнять некоторые действия. В рамках данной книги вы не будете этим заниматься, поэтому в дальнейшем просто не обращайте внимания на эту переменную.

Почему нет переменной **МойВозраст**, вы, наверное, уже догадались. Движок останавливается **перед** тем, как выполнить инструкцию. Инструкция ещё не выполнялась, переменной ещё нет.

Надо сказать движку, чтобы он выполнил вашу инструкцию. Ту, перед которой он остановился.

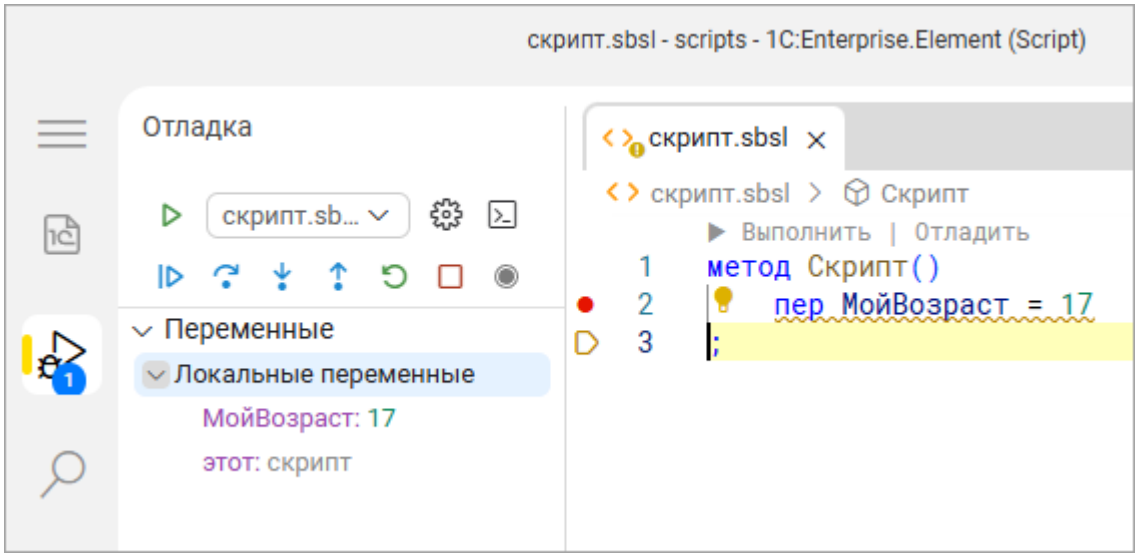
Для выполнения программы по одному шагу (или по нескольким шагам сразу) есть ряд кнопок. Они расположены в командной панели наверху.



Третья из них, **Шаг с заходом**, именно та, что вам нужна. Она используется чаще всего. Если подвести к ней указатель мыши и подержать, то появится подсказка.

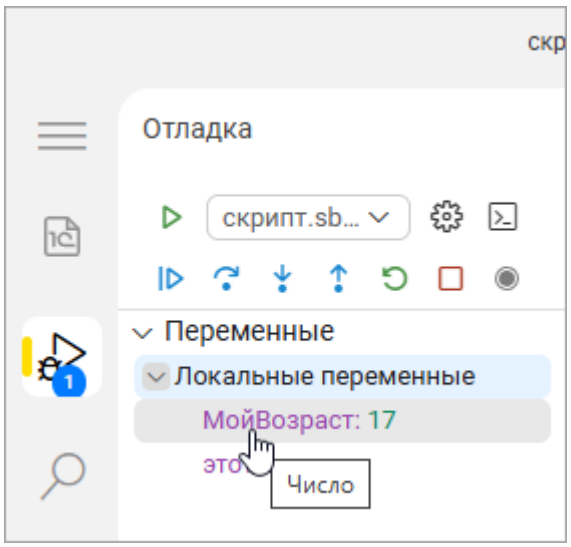
Это же действие можно выполнить из главного меню: **Главное меню > Запустить > Шаг с заходом**. Тут написано, что это действие также можно выполнить клавишей **F11**.

Нажмите **F11** — это самый простой и удобный способ, которым пользуются программисты. Движок выполнит одну инструкцию и остановится перед выполнением следующей.

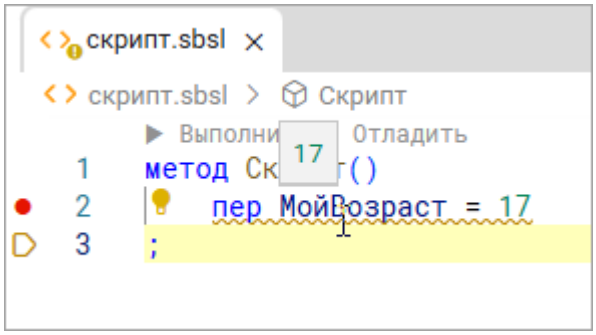


Строка, перед выполнением которой движок остановился, опять выделена жёлтым и отмечена стрелкой. Ваша инструкция объявления переменной выполнилась.

Это действительно так, потому что в представлении **Отладка** в группе **Локальные переменные** появилась переменная **МойВозраст**, имеющая значение **17**. Тип этой переменной — **Число**. Чтобы узнать это, подведите мышь к имени переменной — её тип появится во всплывающей подсказке.



Есть и другой способ посмотреть значение переменной. Вы можете просто подвести курсор мыши к тому месту программы, где написано имя переменной, и подержать немного. Появится подсказка, в которой будет написано значение переменной.



После того как вы посмотрели всё, что вам нужно, вы можете сказать движку, чтобы он завершил отладку. Для этого выполните команду **Остановить**. Это последняя команда в той же командной панели, где расположена команда **Шаг с заходом**.

Это же действие можно выполнить из главного меню: **Главное меню > Запустить > Остановить** или **Shift +F5**.

Выполняя примеры, которые расположены дальше, придерживайтесь этой последовательности действий:

1. Внесите изменения в скрипт (напишите новую команду или измените старую).
2. Установите точку останова. Если точка останова уже стоит в той строке, которая вам нужна, ничего делать не надо.
3. Запустите движок в режиме отладки.
4. Посмотрите значения переменных, сделайте несколько шагов — **F11**.
5. Остановите отладку — **Shift+F5**.

Инструкция присваивания

Теперь вы умеете создавать переменные и помещать в них значения. Познакомьтесь с тем, как можно изменять значения переменных.

Редко бывает так, что программист создал переменную, поместил в неё какое-то значение, а дальше только пользуется этим значением. Чаще бывает по-другому.

Программист создал переменную, поместил в неё значение, использовал тут, там. Потом поместил в неё другое значение. И тоже использовал эту переменную где-то. Потом опять изменил значение переменной, и так далее.

Изменить значение переменной просто — нужно написать инструкцию присваивания.

Например, в переменной **МойВозраст** находится значение **17**. Чтобы изменить значение этой переменной, напишите ещё одну инструкцию и поставьте на ней точку останова. Предыдущую точку останова удалите.

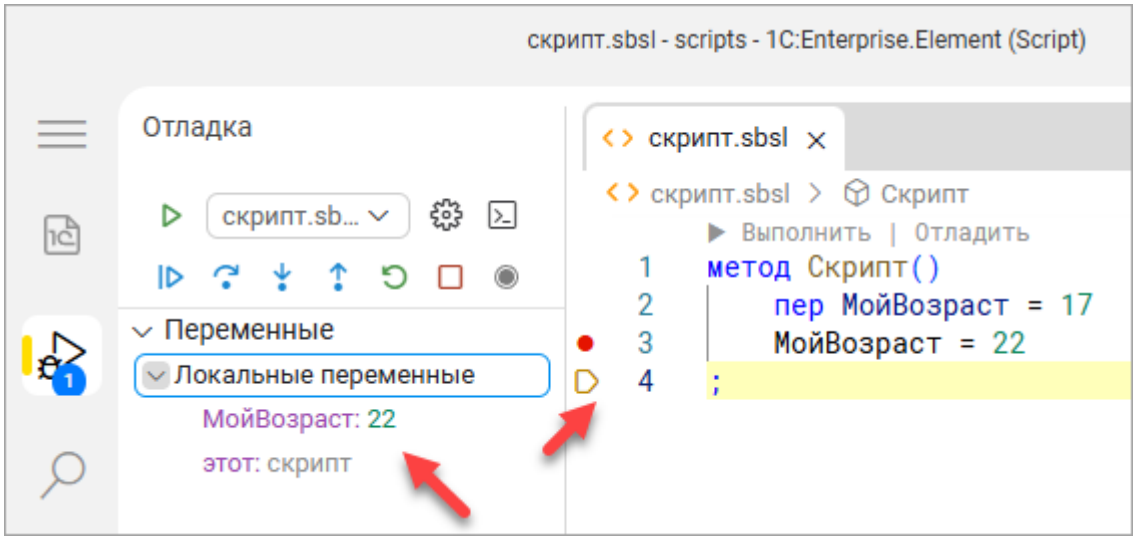
```
метод Скрипт()
    пер МойВозраст = 17
    МойВозраст = 22
;
```

Самая простая и самая популярная инструкция в «1С:Элементе» — это инструкция присваивания. Отличительным признаком инструкции присваивания является знак равенства «=». Это обязательный символ в этой инструкции.

Инструкция присваивания работает следующим образом. Сначала движок «1С:Предприятие.Элемент Скрипт» вычисляет всё, что находится справа от знака равенства. После этого он помещает результат в то место, которое обозначено слева от знака равенства.

Запустите скрипт в режиме отладки и посмотрите, как будет изменяться значение переменной **МойВозраст**.

Сначала оно будет равно **17**. Когда вы сделаете один шаг (**F11**), вторая инструкция выполнится и значение переменной станет **22**.



На этой картинке хорошо видно, что одинаковые на первый взгляд инструкции имеют совершенно разный смысл.

Первая инструкция, объявления переменной, создаёт переменную **МойВозраст** и помещает туда значение **17**. Вторая инструкция, присваивания, ничего не создаёт. В ту переменную, которая уже существует, она помещает значение **22**.

Контекстная подсказка

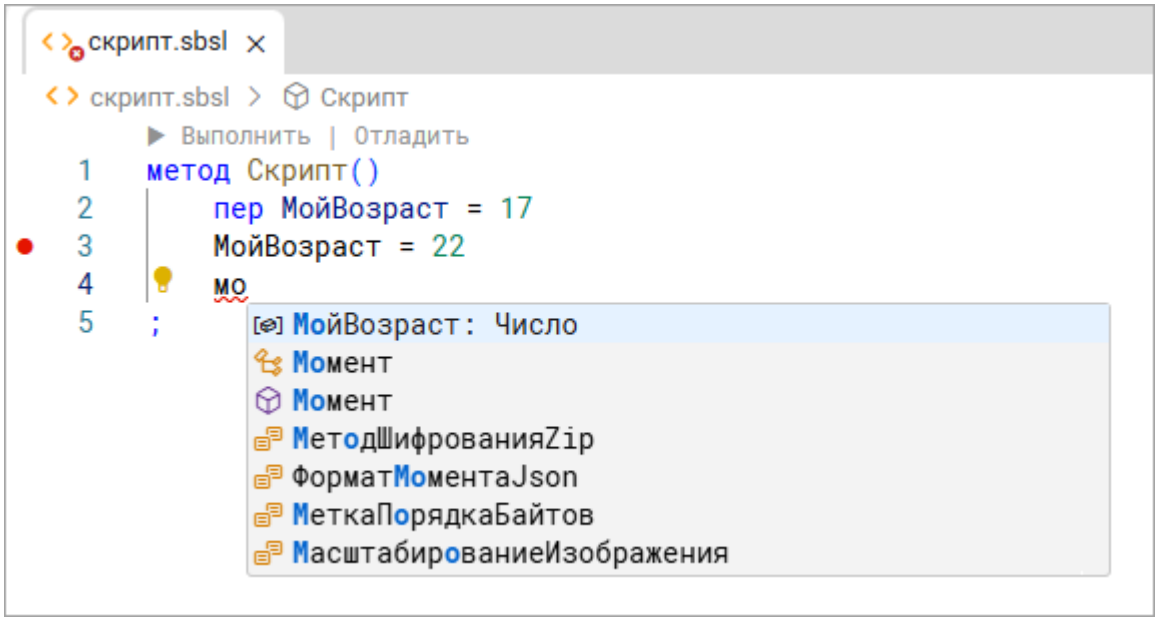
Сейчас подходящее время познакомиться с ещё одним важным инструментом разработчика — *контекстной подсказкой*. Она значительно упрощает жизнь программиста и позволяет избежать большого количества нелепых ошибок. Контекстная подсказка особенно полезна на первом этапе, когда многие слова в «1С:Элементе» являются для вас новыми и непривычными.

Когда вы писали вторую инструкцию присваивания, очень важно было не сделать ошибку в имени переменной **МойВозраст**. Если бы вы написали **МойВозратс** (случайно переставили местами две последние буквы), среда разработки «1С:Предприятие.Элемент» подумала бы, что вы имеете в виду другую переменную. Вместо того, чтобы поместить значение **22** в переменную **МойВозраст**, она показала бы сообщение об ошибке, потому что переменную с таким именем вы не создавали.

Попробуйте написать ещё одну инструкцию присваивания, но уже используя контекстную подсказку.

В переменную **МойВозраст** нужно поместить значение **20**.

В новой строке начните писать **мо**, появится контекстная подсказка.



Список контекстной подсказки будет содержать всё, что среда разработки «знает». Всё, что вы можете использовать в этом месте без каких-либо дополнительных «объяснений». Причём этот список отсортирован таким образом, чтобы сверху были наиболее подходящие предложения. В найденных словах выделены символы, которые вы начали вводить — **мо**.

На первой строке находится переменная **МойВозраст**. Среда разработки тоже «знает» её, потому что эту переменную вы создали парой строк выше. По этой же причине ока подсказывает вам её тип — **Число**.

Вы можете нажать мышью на строку **МойВозраст**, и она подставится в текст скрипта. Вы можете перейти к этой строке с помощью курсора и затем нажать **Ввод** на клавиатуре. Попробуйте.

Также вы можете не искать «глазами» нужную строчку в этом списке. Уже после того, как окно контекстной подсказки открыто, вы можете продолжать набирать имя переменной на клавиатуре — **йвоз**. У вас получится **мойвоз**, а в контекстной подсказке будет единственная строка **МойВозраст: Число**, потому что других подходящих строк больше нет.

Вам останется только нажать клавишу **Ввод**, и **МойВозраст** подставится в текст программы. Попробуйте.

Использовать контекстную подсказку удобно, полезно и нужно. Во-первых, это поможет вам избежать многих орфографических ошибок. Во-вторых, значительно сэкономит ваше время, потому что большинство слов вам не нужно будет дописывать до конца. Среда разработки допишет за вас.

Если вы при открытой контекстной подсказке не стали заканчивать вводимый текст, а перешли на другую строку, контекстная подсказка закроется. Вы всегда можете снова установить курсор в нужное место и вызвать её, нажав **Ctrl+Пробел**.

1. Задание простое

Создайте переменную **МойРост**. Запишите в неё свой рост. Затем измените значение этой переменной на тот рост, который будет у вас, например, через год.

Запустите скрипт в режиме отладки и посмотрите, как будет изменяться значение этой переменной. Решение смотрите здесь *(291)*.

2. Задание сложное

Создайте переменную, в которой находится ваше имя. Посмотрите значение этой переменной в режиме отладки. Решение смотрите здесь *(291)*.

3. Задание сложное

Создайте переменную, в которой находится ваш домашний адрес. Решение смотрите здесь *(291)*.

4. Задание сложное

Выберите любой учебный день. Создайте две переменные. В одной будет храниться название первого урока в этот день, в другой — название второго урока. Решение смотрите здесь *(291)*.

5. Задание сложное

Создайте переменную, в которой будет храниться телефонный номер. Ваш номер или любого другого человека — не важно. Запишите в неё свой телефонный номер. Решение смотрите здесь *(291)*.

6. Задание сложное

Создайте переменную, в которой будет храниться оценка, полученная на уроке. На одном уроке вы получили пятёрку, на другом уроке — четыре с плюсом. Запишите это в программе. Посмотрите в режиме отладки, как будет изменяться значение этой переменной. Решение смотрите здесь *(292)*.

Выражение

Теперь вы знаете, как создавать переменные, как присваивать им значения. Ещё раз посмотрите, как схематически выглядит инструкция присваивания, которой вы сейчас занимаетесь:

```
имя_переменной = выражение
```

Сейчас ваша задача — понять, что это за «выражение», которое находится справа от знака равенства.

Выражение – это математическая, или логическая, или строковая формула, по которой вычисляется значение. Например, если вы занимаетесь пять дней в неделю по шесть занятий каждый день, то количество занятий в неделю можно посчитать, перемножив 6 на 5. **6 * 5** — это выражение.

```
КоличествоЗанятий = 6 * 5
```

Выражение обычно состоит из одной или нескольких *операций*. В этом примере одна операция — это операция умножения. Она обозначается звёздочкой «*».

Значение, над которым выполняется операция, называется *операндом*. В примере два операнда: число **6** и число **5**.

Самое простое выражение может не содержать ни одной операции, а только значение. Такой пример вы уже писали раньше.

```
МойВозраст = 22
```

В этом примере значение — это 22, то есть литерал типа `Число`.

В выражениях можно использовать не только литералы, но и переменные. Например, вы можете взять пример с количеством занятий в неделю и изменить его так, чтобы умножались не два числа, а две переменные.

```
метод Скрипт()
    пер ЗанятийВДень = 6
    пер ДнейВНеделю = 5
    пер КоличествоЗанятий = ЗанятийВДень * ДнейВНеделю
;
```

Совет:

Сразу возьмите себе за правило ставить пробелы перед знаками операций (+, -, *, /) и после них. Тогда текст программы будет читаться легко.

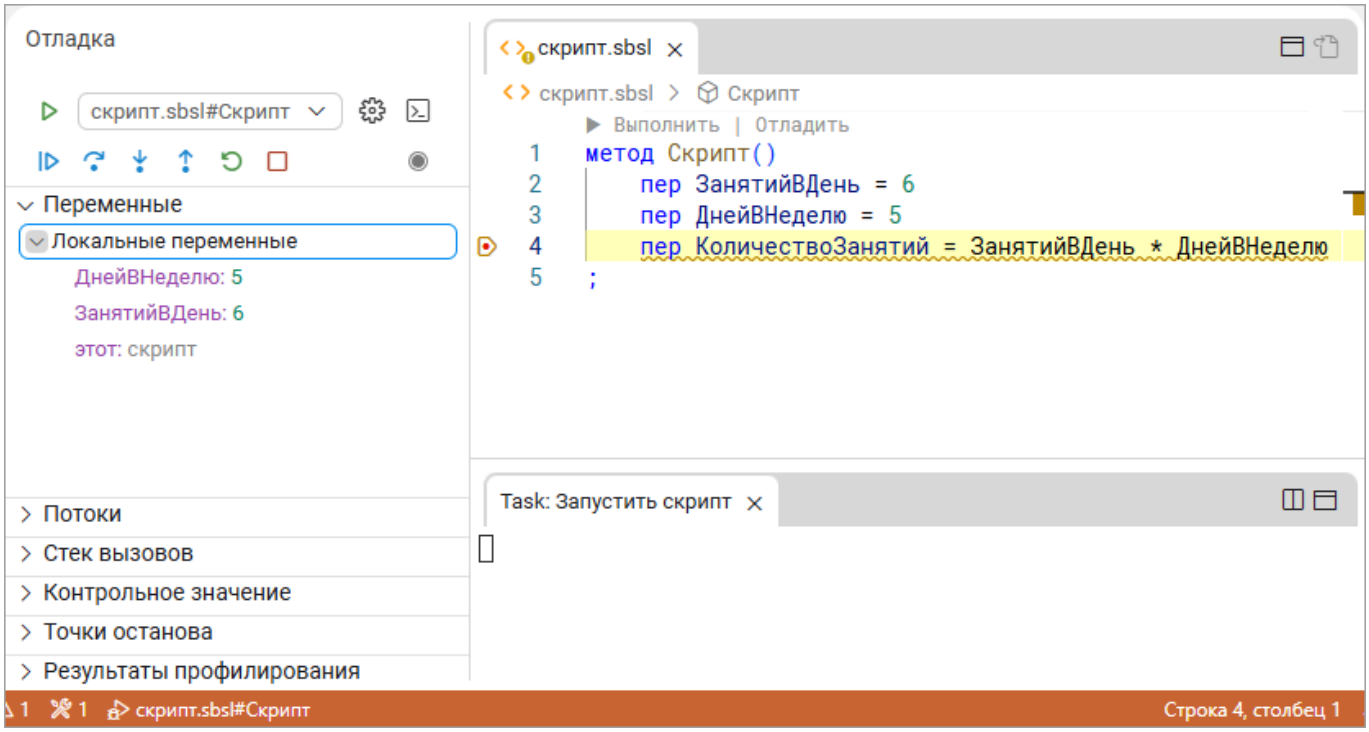
Скопируйте этот пример себе в скрипт. Установите точку останова во второй строке примера и пройдите его по шагам в режиме отладки. Посмотрите, как меняются значения переменных.

Как посмотреть выражение в тексте программы?

Вспомните, как работает инструкция присваивания. Сначала вычисляется выражение справа, и только после этого полученное значение помещается в переменную или свойство, которые написаны слева.

Есть способ посмотреть, какое значение получается справа, ещё до того, как оно будет присвоено переменной слева. Познакомьтесь с ним.

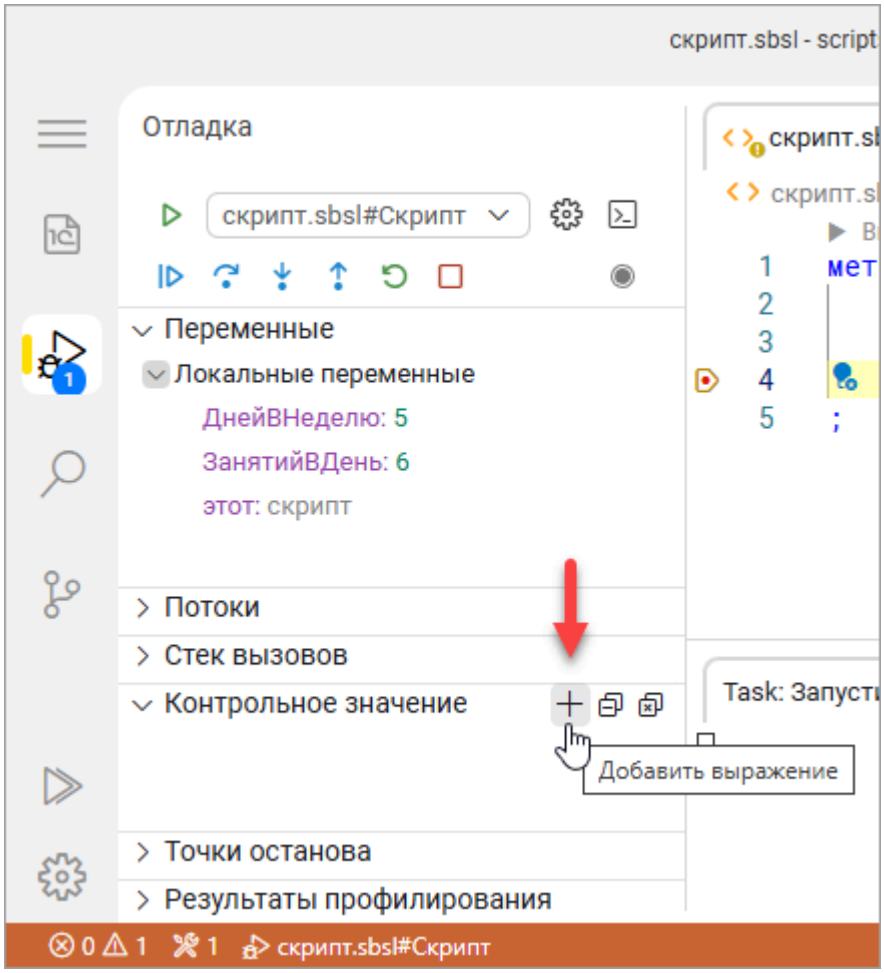
В последнем примере поставьте точку останова на третьей строке и запустите отладку.



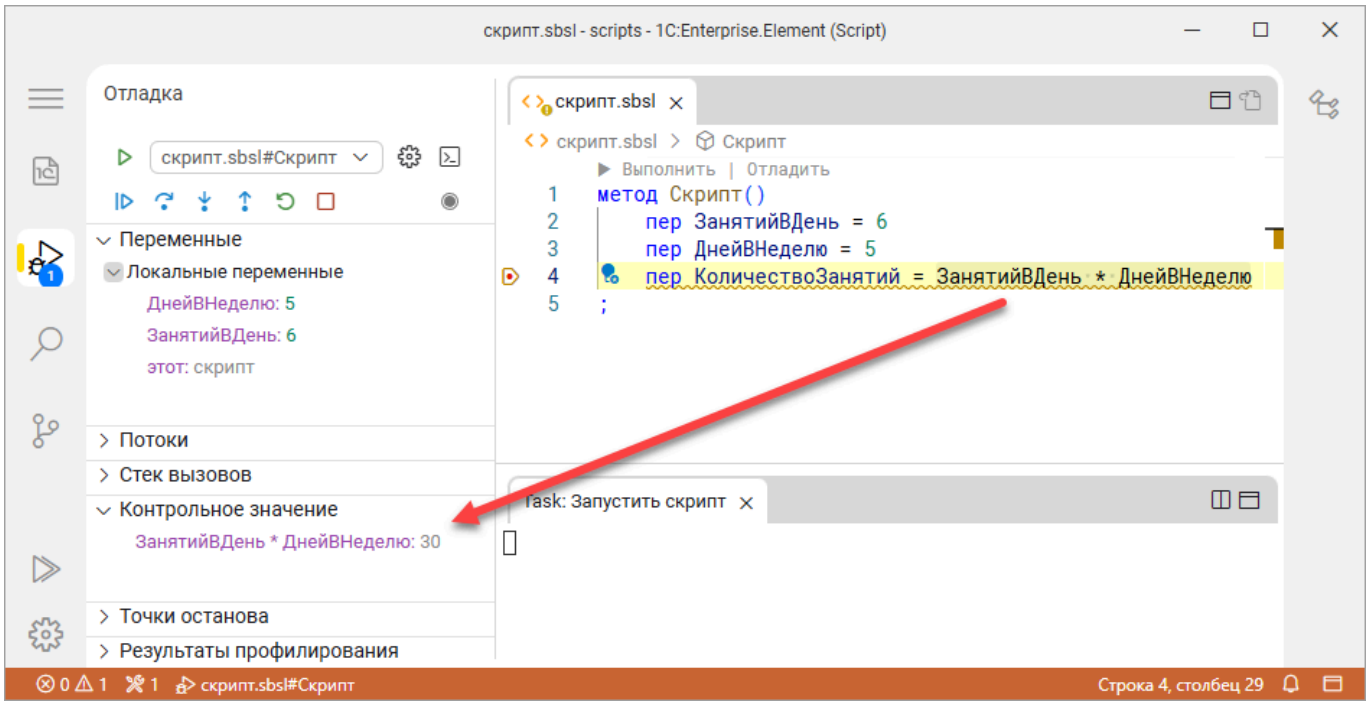
Обратите внимание, что переменная **КоличествоЗанятий** пока не существует. Однако значения переменных **ЗанятийВДень** и **ДнейВНеделю** уже известны.

Теперь с помощью мыши выделите всё выражение, которое находится справа от знака равенства, и нажмите **Ctrl+C**, чтобы скопировать его в буфер обмена.

Раскройте раздел **Контрольное значение** и нажмите **Добавить выражение**.



Вставьте содержимое буфера обмена (**Ctrl+V**) и нажмите **ОК**.



В разделе **Контрольное значение** вы увидите результат вычисления выражения.

7. Задание простое

В одной переменной сохраните вашу среднюю скорость — 5 км/ч. В другой переменной сохраните расстояние до школы — 6 км. В третьей переменной посчитайте количество минут, которое вам понадобится, чтобы дойти до школы. Решение смотрите здесь *(292)*.

8. Задание сложное

Используйте среднюю скорость и расстояние до школы из задания 7. В третьей переменной посчитайте, сколько раз за сутки вы сможете дойти до школы и вернуться обратно, если не будете останавливаться и спать. После этого с помощью вычисления выражения посмотрите, сколько километров вы можете пройти за сутки. Решение смотрите здесь *(292)*.

9. Задание сложное

В этом же примере (8) посчитайте, сколько раз вы сможете сделать то же самое, если будете ехать на велосипеде только в светлое время суток.

Средняя скорость велосипедиста 15 км/ч. Светлое время суток длится в среднем 13 часов. Решение смотрите здесь *(292)*.

10. Задание сложное

В вашем портфеле были только учебники. В понедельник, чтобы перекусить в школе, вы взяли из дома 2 яблока. Но не стали их есть, и они остались в портфеле. Во вторник и в среду вы тоже брали яблоки из дома и оставляли их в портфеле. Сколько яблок будет в вашем портфеле в среду, если каждый день вы брали из дома на два яблока больше, чем в предыдущий?

Для решения этой задачи используйте две переменные: **ВПортфеле** (количество яблок в портфеле) и **ВзялИзДома** (количество яблок, которое вы взяли из дома). Решение смотрите здесь *(293)*.

Арифметические операции

Арифметические операции

Если ваши значения имеют тип `Число`, то с ними можно выполнять арифметические операции: сложение, вычитание, умножение и деление. Эти операции вам хорошо известны. Единственное, что может вызвать трудность, это каким знаком обозначать умножение и деление.

Но и это вы уже знаете. Умножение обозначается звёздочкой «*», а деление — косой чертой «/».

Кроме этого есть и другие операции над числами, о которых вы можете прочитать в руководстве разработчика:

- получение остатка от деления, обозначается процентом «%»;
- возведение в степень, обозначается двумя звёздочками «**»;
- изменение знака, обозначается минусом «-».

Одновременно в выражении может участвовать любое количество арифметических операций. Но выполняются они не в той последовательности, как они написаны. Вам это известно из математики. В

математике существует определённый порядок выполнения операций. А кроме этого используются скобки, чтобы указать именно тот порядок, который вам нужен.

В «1С:Элементе» то же самое:

- Если в выражении есть скобки, сначала вычисляется то, что в скобках.
- Если скобок нет, то в первую очередь выполняются операции умножения и деления в том порядке, в котором они написаны.
- В самую последнюю очередь выполняются операции сложения и вычитания (в том порядке, в котором они написаны).

Инструкции присваивания, совмещённые с арифметическими операциями

Движок «1С:Предприятие.Элемент Скрипт», так же как и вы, сначала вычисляет выражение, а потом помещает его в переменную слева. Поэтому переменную, которая находится слева от знака равенства, можно использовать в выражении в этой же инструкции присваивания.

Попробуйте сделать такой пример. Сначала в переменную **МойВозраст** запишите свой возраст. А теперь представьте, что прошёл один год и в следующей строке вам нужно записать в эту переменную свой возраст через год. Как вы это сделаете?

Ответ находится ниже. Если у вас получилось то же самое — замечательно! Если нет — не расстраивайтесь, скопируйте пример к себе в скрипт.

```
метод Скрипт()
    пер МойВозраст = 17
    МойВозраст = МойВозраст + 1
;
```

На первый взгляд такая конструкция кажется странной. Но только не для компьютера.

Установите точку останова на второй строке примера и в режиме отладки посмотрите, как изменится значение переменной **МойВозраст**. Сначала в ней будет значение **17**. После выполнения третьей строки значение изменится на **18**.

Инструкция **МойВозраст = МойВозраст + 1** выглядит громоздко. Её можно написать лаконичнее, для этого в языке есть инструкции, совмещённые с арифметическими операциями.

Например, если к значению переменной нужно прибавить что-то и поместить в эту же переменную, можно использовать операцию `+=`. Тогда возможны два равноправных способа записи:

```
// Инструкция присваивания
МойВозраст = МойВозраст + 1

// Инструкция сложения с присваиванием
МойВозраст += 1
```

Аналогичные инструкции существуют для вычитания, умножения и деления.

Вычитание с присваиванием. Оставшееся время уменьшится на 60 единиц, например, секунд.

```
ОставшеесяВремя -= 60
```

Умножение с присваиванием. Количество занятий увеличится в два раза.

```
КоличествоЗанятий *= 2
```

Деление с присваиванием. Количество апельсинов уменьшится в четыре раза.

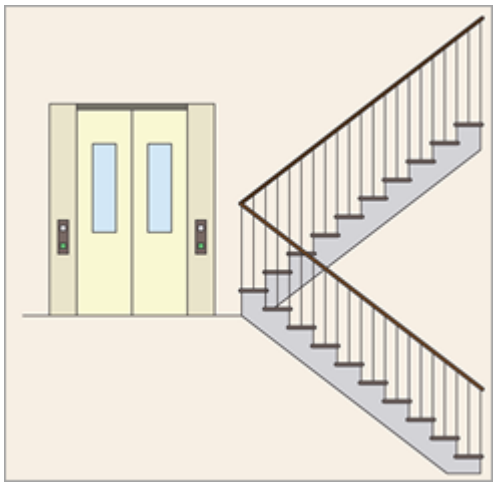
```
КоличествоАпельсинов /= 4
```

11. Задание простое

Создайте четыре переменные: **ДлинаУрока** — 45 минут; **ДлинаПеремены** — 15 минут; **ДлинаБольшойПеремены** — 25 минут; **ВсегоУроков** — 6. Посчитайте, сколько минут вы проводите в школе в течение дня, если одна из перемен между уроками всегда большая. Решение смотрите здесь *(293)*.

12. Задание сложное

Лестницы в многоэтажных домах обычно устроены следующим образом.



Чтобы подняться с этажа на этаж, нужно пройти два лестничных марша. Ещё несколько ступеней есть перед входом в подъезд.

Сколько ступеней нужно пройти, чтобы подняться на ваш этаж?

Предусмотрите, что у разных людей в разных домах следующие величины могут быть разными. Перед вычислением сохраните эти значения в отдельных переменных:

- количество ступеней перед входом в подъезд;
- количество ступеней в марше;
- этаж.

Решение смотрите здесь *(293)*.

> **Примечание:**
>
> Работа с числами на углублённом уровне описана здесь *(157)*.

Операции со строками

Если ваши значения имеют тип `Строка`, то они тоже могут участвовать в выражениях. Операций, которые часто используются при работе со строками, всего две. Сейчас вы рассмотрите одну из них.

Эта операция заключается в том, чтобы к одной строке дописать другую — «склеить» строки. Называется она сложным словом *конкатенация*, а обозначается плюсом «+».

> **Примечание:**
>
> Другая частая операция — это сравнение строк на равенство, её вы рассмотрите позже *(52)*.

Сделайте такой пример. В одной переменной сохраните название вашего города. В другой переменной — название улицы, на которой вы живёте. А в третью переменную с помощью выражения запишите ваш адрес, который будет состоять из названия города и улицы.

Чтобы проверить себя, посмотрите на пример.

```
метод Скрипт()
    пер Город = "Красноярск"
    пер Улица = "Цветочная"
    пер Адрес = Город + Улица
;
```

Установите точку останова на последней строке и в режиме отладки посмотрите, какое значение будет в переменной **Адрес**.

Вы увидите, что две строки действительно «склеились» и получилась строка **КрасноярскЦветочная**. С одной стороны, это хорошо, потому что конкатенация работает. С другой стороны, не очень хорошо, потому что результат выглядит «не очень».

Чтобы получить хороший результат при конкатенации, часто используют литералы типа `Строка`, чтобы отделить одно строковое значение от другого. В вашем примере хочется вместо **КрасноярскЦветочная** видеть **Красноярск, Цветочная**.

Чтобы добиться такого результата, измените последнюю строку так, как показано в примере.

```
метод Скрипт()
    пер Город = "Красноярск"
    пер Улица = "Цветочная"
    пер Адрес = Город + ", " + Улица
;
```

Запустите этот пример в режиме отладки и посмотрите, какой получается результат.

13. Задание простое

Используя переменную **Возраст** типа `Строка`, запишите фразу «Мой возраст 17 лет». Решение смотрите здесь *(293)*.

14. Задание сложное

При отправке почтовых сообщений существует определённый порядок перечисления реквизитов адреса:

1. Название улицы, номер дома, номер квартиры.
2. Название населённого пункта (города, посёлка и т. п.).
3. Название района.
4. Название республики, края, области, автономного округа (области).

Используя переменные **Улица**, **НомерДома**, **НомерКвартиры**, **Город** и **Область**, запишите адрес: Можайское ш., д. 37, кв. 36, г. Одинцово, Московская обл. Решение смотрите здесь *(294)*.

> **Примечание:**
>
> Работа со строками на углублённом уровне описана здесь *(166)*.

Тип «ДатаВремя» и операции с датами

Общая информация

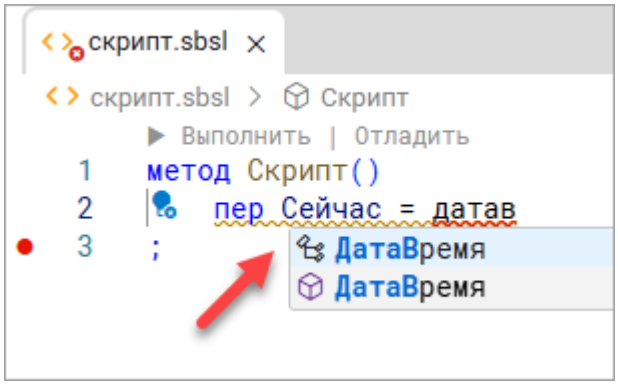
`ДатаВремя` — это первый необычный тип данных, с которым вам придётся столкнуться. В повседневной жизни вы используете даты очень часто и часто делаете с ними какие-то операции. Но в жизни вы привыкли воспринимать даты отдельными частями.

Например, когда вас спрашивают: «Сколько вам лет?» — вы из одного года вычитаете другой год. То есть из одного числа вычитаете другое. Когда вас спрашивают, через сколько минут начнётся фильм, вы тоже из одного числа (количество минут) вычитаете другое число (количество минут).

В «1С:Элементе» не так. В «1С:Элементе» значение типа `ДатаВремя` — это всегда календарная дата вместе со временем с точностью до секунд. Например, **31.03.2016 12:25:59**. Это значит «двенадцать часов двадцать пять минут пятьдесят девять секунд тридцать первого марта две тысячи шестнадцатого года».

Чтобы убедиться в этом, вы можете выполнить простой пример. Напишите в модуле инструкцию, которая показана ниже. Когда будете писать **ДатаВремя.Сейчас()**, не забывайте пользоваться контекстной подсказкой.

В подсказке будет две строки **ДатаВремя** — выбирайте первую, это тип. Вторая строка — для ввода литерала.




```
метод Скрипт()
    пер Сейчас = ДатаВремя.Сейчас()
;
```

Установите точку останова на последней строке, запустите скрипт в режиме отладки и посмотрите значение переменной **Сейчас**. В ней будет текущее время, установленное на вашем компьютере.

Значения типа `ДатаВремя` в «1С:Элементе» представляются в международном формате ISO 8601. Например, **2025-06-13T13:15:50.245** означает 13 часов 15 минут 50 секунд 245 миллисекунд 13 июня 2025 года.

Литерал

Значения типа `ДатаВремя`, так же как числа и строки, можно записывать прямо в тексте скрипта. Это используется не очень часто, но используется. Поэтому посмотрите, как выглядит литерал типа `ДатаВремя`:

```
метод Скрипт()
    пер НачалоЗанятийСегодня = ДатаВремя{2025-05-16 09:00:00}
;
```

> **Примечание:**
>
> Фигурные скобки «{» и «}» можно вводить с помощью Alt-ввода *(227)*.

Конструктор

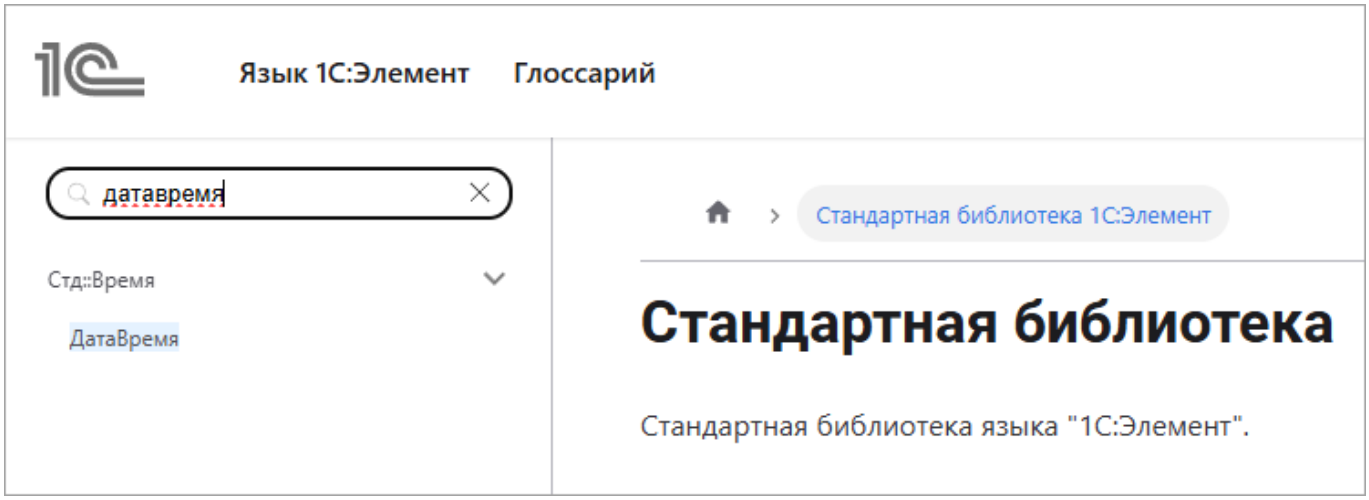
Помимо литерала есть и другой способ записать значение типа `ДатаВремя` в тексте скрипта. Для этого можно использовать методы этого типа.

Один из методов вы уже использовали, когда написали **ДатаВремя.Сейчас()** — это метод `ДатаВремя.Сейчас()`.

Что такое методы, вы узнаете позже *(101)*. Сейчас просто запомните, что это такая «функция», которая записывается через точку «.» от значения или от имени типа.

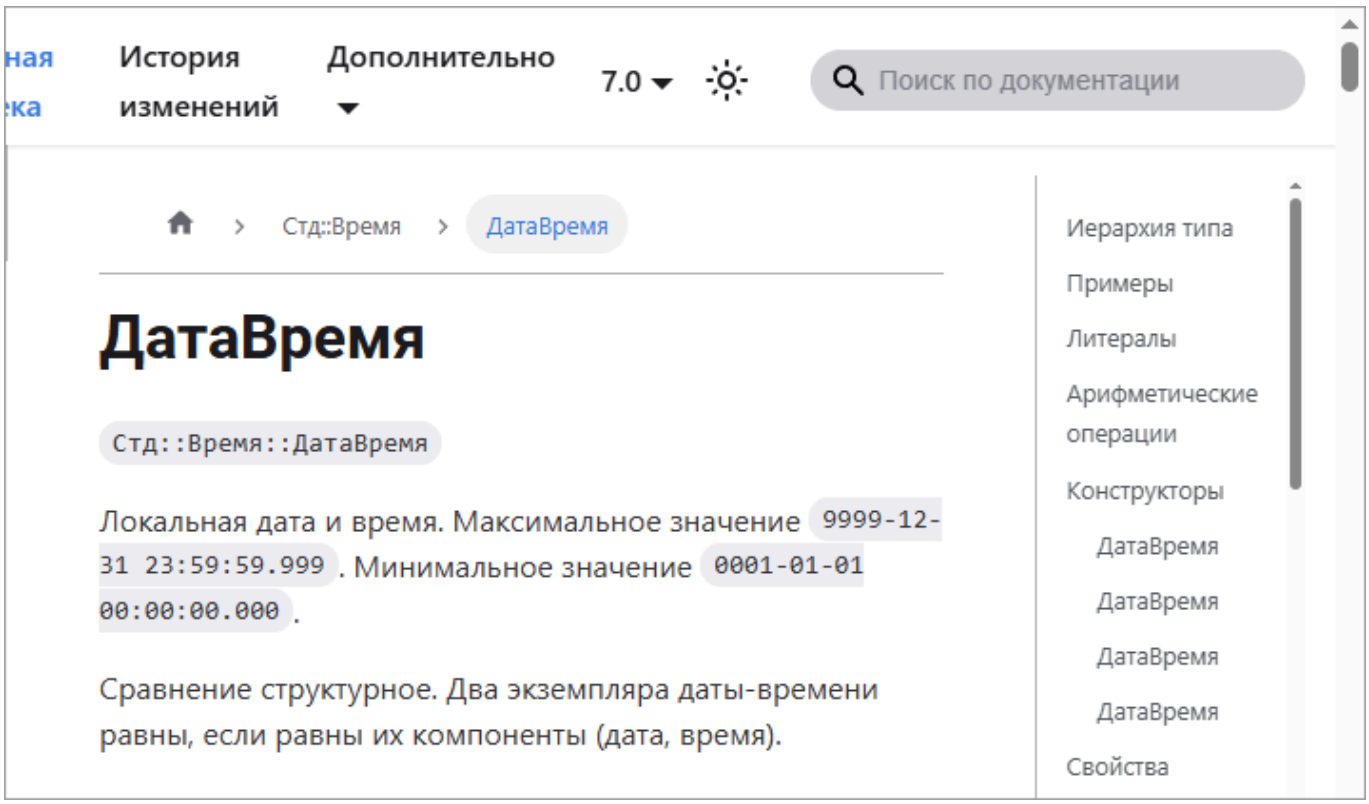
Посмотреть, какие есть методы для работы со значениями типа `ДатаВремя`, очень просто. Для этого вам понадобится справка по стандартной библиотеке, о которой вы уже слышали раньше. Чтобы её открыть, нажмите **Главное меню > Справка > Документация** или перейдите по этой ссылке.

В документации перейдите в раздел **Стандартная библиотека - Справочник по объектной модели**. Слева перечислены типы и пространства имён, над ними есть строка поиска. Введите в неё **ДатаВремя** и нажмите **Ввод**.



Нажмите на найденный тип. Справа вы увидите разделы описаний, в том числе раздел **Методы**. В нём перечислены все методы, которые есть у значений типа `ДатаВремя`. Здесь вас будет интересовать ещё одно незнакомое пока слово — конструкторы. Сейчас просто запомните, что это такие специальные функции, которые позволяют создать значение этого типа из чего-то другого. Например, из чисел, обозначающих год, месяц, день и т. д.

Как видите, их там не один, а четыре, все они называются одинаково. Чем же они отличаются?



Чтобы это понять, нужно кликнуть на один из них. Например, на первый.

Конструкторы

ДатаВремя

```
ДатаВремя(Представление: Строка)
```

Перегрузка:
ДатаВремя(Год: Число, Месяц: Число, День: Число)
ДатаВремя(Год: Число, Месяц: Число, День: Число, Час: Число, Минута: Число, Секунда: Число, Миллисекунда: Число = 0)
ДатаВремя(Дата: Дата, Время: Время)

Преобразует строковое представление `Представление` в соответствующий экземпляр даты-времени.

Исключения

ИсключениеНедопустимыйФормат - если представление не является допустимым.

Забегая вперёд, конструктор, когда вы его напишете в скрипте, будет выглядеть следующим образом:

```
пер НачалоЗанятийСегодня = новый ДатаВремя(...)
```

Любой конструктор всегда начинается с ключевого слова `новый`, за ним через пробел идёт то, что написано в справке по стандартной библиотеке: имя метода, а в скобках — тот или иной набор аргументов.

Так вот, эти четыре конструктора отличаются количеством и типом своих аргументов. И в описании каждого из них, чтобы вам было проще, в разделе **Перегрузка** перечислены другие конструкторы со своими аргументами.

В частности, вы видите, что есть конструктор с семью аргументами, все они имеют тип `Число`. Можете нажать на эту ссылку или пролистать вниз до описания этого конструктора.

ДатаВремя

```
ДатаВремя(
Год: Число,
Месяц: Число,
День: Число,
Час: Число,
Минута: Число,
Секунда: Число,
Миллисекунда: Число = 0)
```

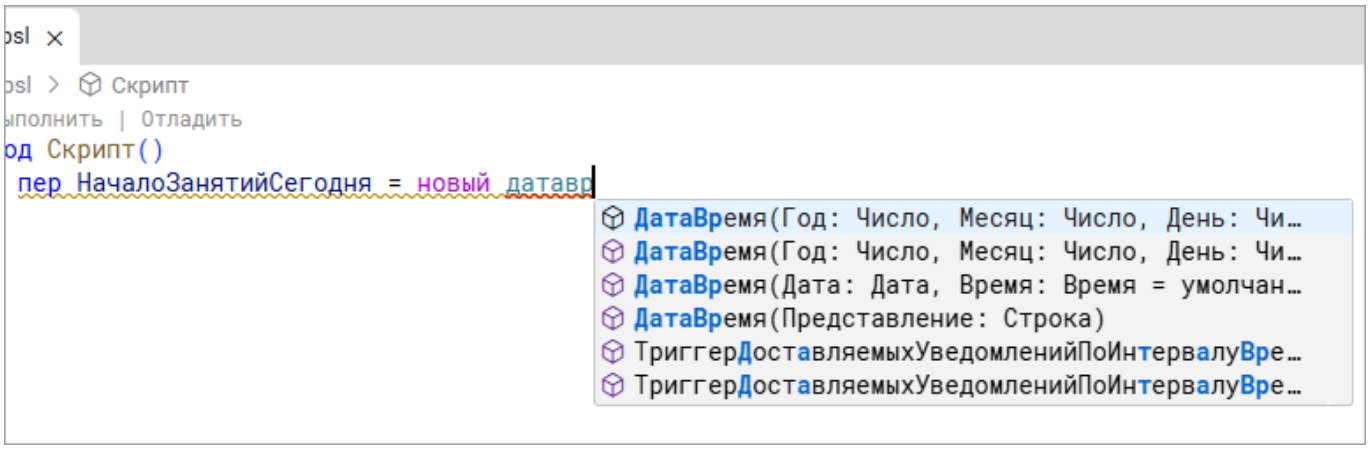
Перегрузка:
ДатаВремя(Представление: Строка)
ДатаВремя(Год: Число, Месяц: Число, День: Число)
ДатаВремя(Дата: Дата, Время: Время)

Создает дату-время на основе переданных компонентов `Год`, `Месяц`, `День`, `Час`, `Минута`, `Секунда`, `Миллисекунда`.

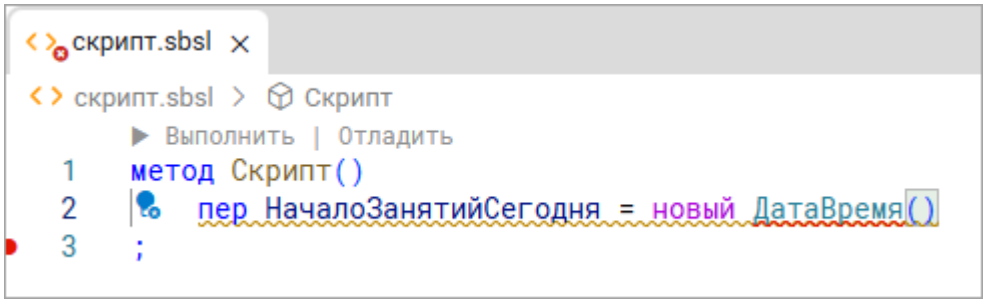
Как видите, он создаёт значение типа `ДатаВремя` из семи числовых компонентов.

Попробуйте использовать этот конструктор. Вернитесь в свой скрипт, напишите **пер НачалоЗанятийСегодня =**.

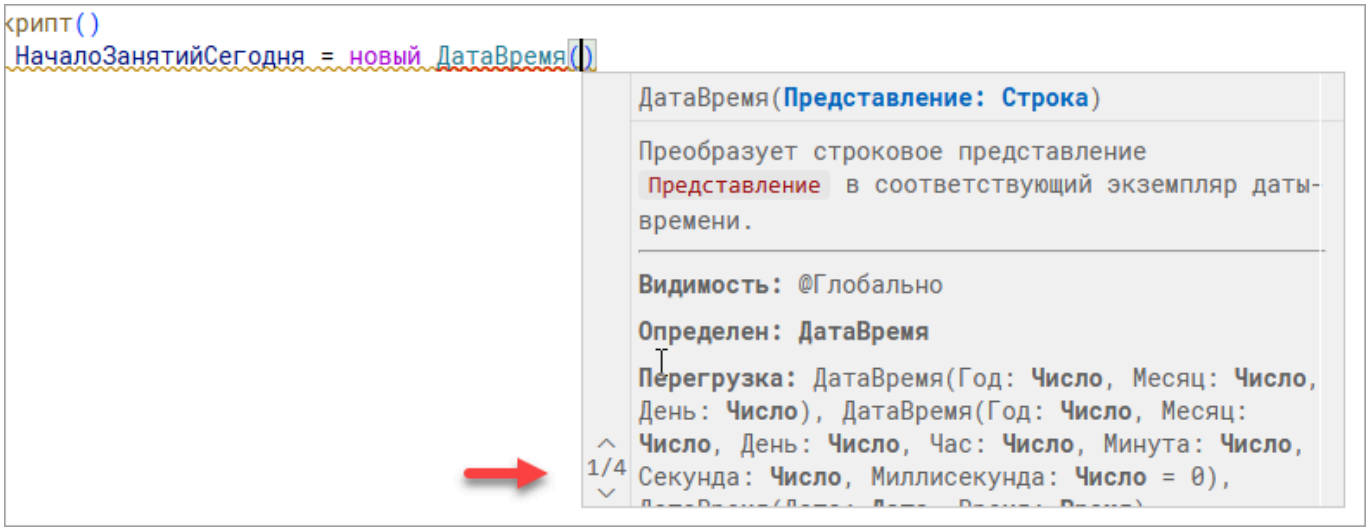
После знака равенства напишите **новый датавр**.



Контекстная подсказка покажет вам все четыре конструктора. Нажмите на любой из них. Контекстная подсказка допишет за вас конструктор и закроется.

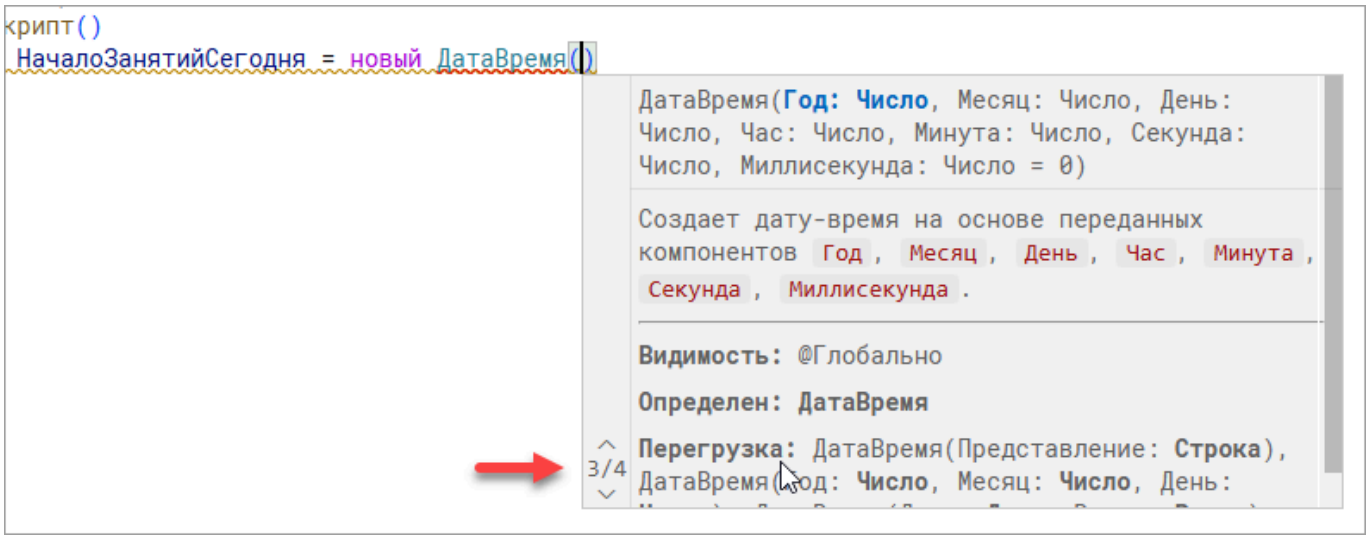


Теперь нажмите **Ctrl+Shift+Пробел**, чтобы открыть контекстную подсказку по аргументам этого конструктора.

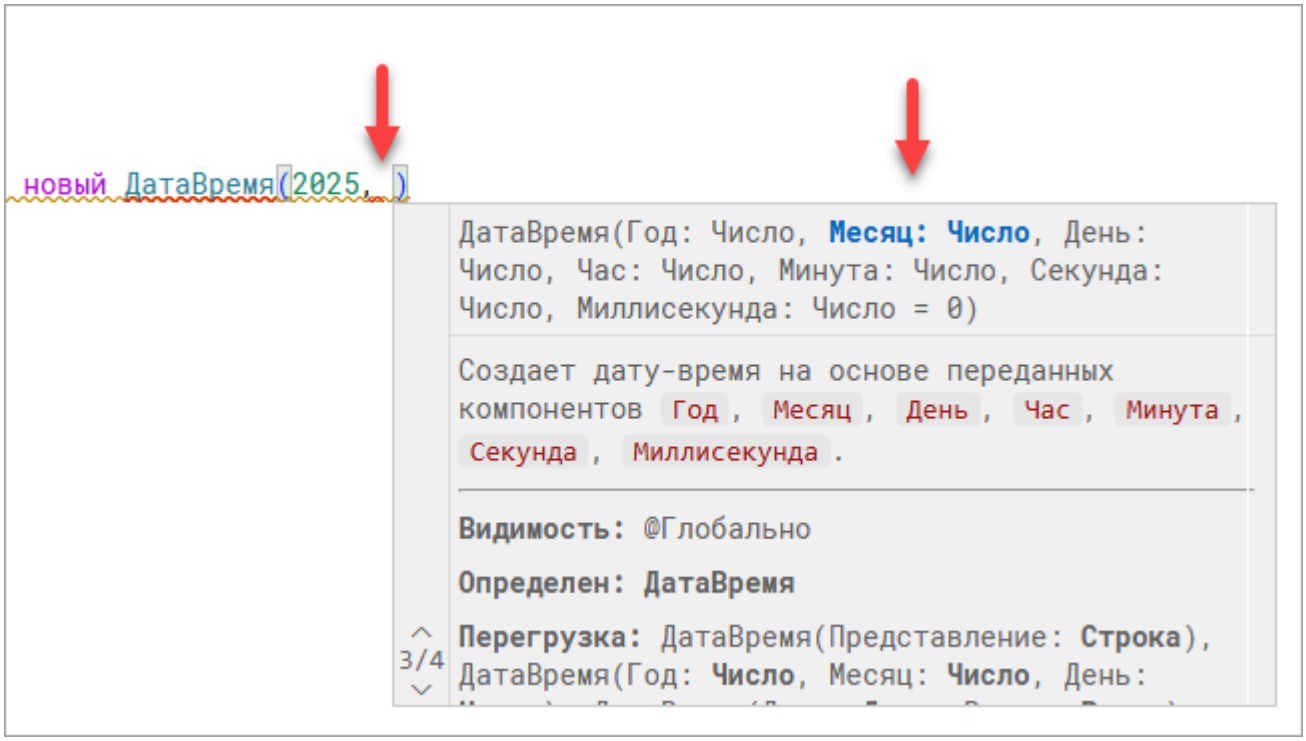


Она подсказывает вам, какие аргументы нужно вводить. В частности, сейчас она подсказывает, что нужно ввести аргумент типа `Число`. Но, к сожалению, это не тот вариант конструктора с семью аргументами, который вам нужен.

Если вы посмотрите в левый нижний угол подсказки, то увидите, что она показывает вам первый вариант конструктора из имеющихся четырёх. Поэтому пролистайте список клавишами **Вверх** или **Вниз** и найдите нужный вам конструктор.



Теперь вы можете вводить оставшиеся аргументы. По мере того, как вы будете ставить запятые, подсказка будет переключаться на следующий аргумент.



А если вы случайно закроете её, то, чтобы снова открыть подсказку по аргументам конструктора (или любого другого метода), поставьте курсор внутри скобок и нажмите **Ctrl+Shift+Пробел**.

Сразу возьмите себе за правило ставить пробелы после запятой. Тогда текст скрипта будет читаться легко.

```
метод Скрипт()
    пер НачалоЗанятийСегодня = новый ДатаВремя(2025, 06, 13, 13, 15, 50)
;
```

Вот так, пользуясь контекстной подсказкой, допишите инструкцию до конца. Запустите скрипт в режиме отладки и убедитесь, что все работает правильно.

Какой вариант написания в тексте скрипта значения типа `ДатаВремя` выбрать: с помощью литерала или с помощью конструктора? Всё зависит от конкретной ситуации. Если вы получаете значения года, месяца, дня и т. д. из другого скрипта в виде чисел, то лучше использовать конструктор. Если вы точно заранее знаете, какая `ДатаВремя` вам нужна (как в этом случае), то тогда «1С:Элемент» рекомендует использовать литерал.

Начало и конец дня

Порой вам могут понадобиться такие даты, как начало некоторого дня и конец некоторого дня. Для их получения существуют два метода: `ДатаВремя.НачалоДня()` и `ДатаВремя.КонецДня()`.

Познакомьтесь с тем, где в «1С:Элементе» день начинается и где он заканчивается. Верните свой скрипт к тому виду, когда вы получали текущую дату. Назовите переменную **НачалоСегодня**.

```
метод  Скрипт()
      пер  НачалоСегодня  =  ДатаВремя.Сейчас()
;
```

ДатаВремя.Сейчас() — это выражение, которое после вычисления имеет значение типа `ДатаВремя`. Значит вы можете после этого выражения поставить точку и вызвать метод `ДатаВремя.НачалоДня()`, который есть у этого типа. Контекстная подсказка вам поможет по мере того, как вы будете набирать **началод...**

```
метод  Скрипт()
      пер  НачалоСегодня  =  ДатаВремя.Сейчас().НачалоДня()
;
```

Запустите скрипт в режиме отладки и посмотрите, чему равно **НачалоСегодня**. Это будет сегодняшняя дата и нулевое время. Например, **2025-06-13T00:00:00.000**.

Теперь посмотрите, чему равен конец дня. Создайте другую переменную, в которую будет помещён результат метода `ДатаВремя.КонецДня()`. Только не повторяйте предыдущую строку, а возьмите переменную **НачалоСегодня**. Ведь в ней некоторое значение из сегодняшнего дня, значит, от него можно получить конец сегодняшнего дня.

```
метод  Скрипт()
      пер  НачалоСегодня  =  ДатаВремя.Сейчас().НачалоДня()
      пер  КонецСегодня  =  НачалоСегодня.КонецДня()
;
```

Запустите «1С:Элемент» в режиме отладки и посмотрите, чему равно **КонецСегодня**. Это будет сегодняшняя дата и время 23:59:59:999. Например, **2025-06-13T23:59:59.999**.

Сложение и вычитание, тип «Длительность»

Теперь познакомьтесь с тем, какие операции можно выполнять со значениями типа **ДатаВремя**.

Сложение с «Длительностью»

В «1С:Элементе» существует тип `Длительность`. Он предназначен для работы с интервалами времени. Этот тип хранит количество миллисекунд, описывающих некоторый интервал времени.

Значение типа `Длительность` можно записать литералом, причём в очень удобном, легко читаемом виде. Например:

```
5д14ч30м
```

Эта запись означает 5 дней, 14 часов и 30 минут. Также можно использовать **с** для секунд и **мс** для миллисекунд.

Так вот, чтобы прибавить к значению `ДатаВремя` произвольный интервал, можно воспользоваться сложением со значением типа `Длительность`.

```
метод Скрипт()
    пер НаКакоеТоВремяБольше = ДатаВремя.Сейчас() + 1ч23м40с
;
```

Добавление с помощью методов

Если интервал времени, который вы хотите добавить, измеряется в часах, годах, днях, минутах и т. д., то можно использовать метод `ДатаВремя.ДобавитьЧасы()` и ему подобные. В качестве аргумента нужно передать количество добавляемых единиц — например, один час.

```
метод Скрипт()
    пер НаЧасБольше = ДатаВремя.Сейчас().ДобавитьЧасы(1)
;
```

Вычитание «Длительности»

Из значения типа `ДатаВремя` можно вычитать значение типа `Длительность`. Помимо очевидного назначения такую операцию можно использовать для того, чтобы получить конец предыдущего дня. Для этого вы получаете начало сегодняшнего дня и вычитаете из него одну миллисекунду.

```
метод Скрипт()
    пер КонецПредыдущегоДня = ДатаВремя.Сейчас().НачалоДня() - 1мс
;
```

Аналогичным образом получается начало следующего дня: нужно получить конец сегодняшнего дня и прибавить к нему одну миллисекунду.

Вычитание «ДатыВремени»

Из значения типа `ДатаВремя` можно вычесть другое значение типа `ДатаВремя`. В результате вы получите значение типа `Длительность`. Оно даже без дополнительных усилий имеет вполне понятное представление, поделённое на часы, минуты и секунды.

Например, интервал времени от начала сегодняшнего дня до его конца можно получить так:

```
метод Скрипт()
    пер НачалоСегодня = ДатаВремя.Сейчас().НачалоДня()
    пер КонецСегодня = ДатаВремя.Сейчас().КонецДня()
    пер Интервал = КонецСегодня - НачалоСегодня
;
```

В результате в переменной **Интервал** будет значение **23:59:59.999**.

Если хочется, можно легко облагородить это значение, получив от него составные части с помощью свойств `Длительность.Часы`, `Длительность.Минуты` и `Длительность.Секунды`.

```
метод Скрипт()
    пер НачалоСегодня = ДатаВремя.Сейчас().НачалоДня()
    пер КонецСегодня = ДатаВремя.Сейчас().КонецДня()
    пер Интервал = КонецСегодня - НачалоСегодня
    пер Сообщение = "Прошло " + Интервал.Часы + " часа, " + Интервал.Минуты + " минут, " + Интервал.Секунды +
;
```

В результате в переменной **Интервал** будет значение **"Прошло 23 часа, 59 минут, 59 секунд"**.

Замена компонентов «ДатыВремени»

Поскольку значение типа `ДатаВремя` состоит из отдельных компонентов (год, месяц, день и т. д.), можно в таком значении заменить любой компонент на другой.

Когда это может понадобиться? Например, у вас запланировано занятие на сегодня, на 16 часов. Но учитель сообщил, что сегодня он занят и занятие переносится на 21-е число, на то же время. Вам нужно в вашем значении `ДатаВремя` заменить число, сохранив все другие компоненты: год, месяц, время.

Для этого используйте метод `ДатаВремя.СДнем()`:

```
метод Скрипт()
    пер ЗанятиеСегодня = ДатаВремя{2025-09-02 16:00:00}
    пер НовоеЗанятие = ЗанятиеСегодня.СДнем(21)
;
```

В результате в переменной **НовоеЗанятие** будет значение **2025-09-21T16:00:00.000**.

15. Задание простое

Запишите начало дня 2 сентября 2025 года с помощью литерала типа `ДатаВремя`. Решение смотрите здесь *(294)*.

16. Задание простое

Запишите начало дня 2 сентября 2025 года с помощью конструктора типа `ДатаВремя`. Решение смотрите здесь *(294)*.

17. Задание простое

В одной переменной сохраните произвольную ДатуВремя. В другой переменной вычислите начало следующего понедельника для неё. Решение смотрите здесь *(294)*.

18. Задание сложное

В одной переменной сохраните произвольную ДатуВремя. В другой переменной вычислите девять утра для неё. Решение смотрите здесь *(295)*.

> **Примечание:**
>
> Работа с датой и временем на углублённом уровне описана здесь *(181)*.

Тип «Булево» и логические операции

Настало время познакомиться с логическими операциями и новым типом данных — `Булево`.

Логические операции

Самая простая логическая операция — это операция **Сравнение на равенство**. Она выясняет, равны между собой два значения или нет.

Эта операция определена для значений любых типов: для чисел, для строк, для дат и так далее. Важно лишь, чтобы оба значения, которые вы сравниваете, были одного и того же типа. Например, два числа или две строки.

Логическая операция **Сравнение на равенство** обозначается двумя знаками равенства «==». В подавляющем большинстве случаев логические операции используются в инструкциях `если`, `для` и `пока`. Но эти инструкции вам ещё незнакомы.

Поэтому знакомиться с операцией **Сравнение на равенство** вы будете с помощью инструкции присваивания. В ней эта операция будет выглядеть немного странно, поэтому для пущей важности вы заключите её в скобки.

```
метод Скрипт()
    пер Результат = ()
;
```

Теперь внутри скобок напишите операцию **Сравнение на равенство**. Например, **5 = 5**.

Затем напишите ещё одну инструкцию присваивания, но так, чтобы в операции **Сравнение на равенство** участвовали разные числа. Например, **5** и **2**.

Запустите скрипт в режиме отладки и посмотрите, чему равны переменные в этих инструкциях.

```
метод Скрипт()
    пер Результат = (5 == 5)
    пер ДругойРезультат = (5 == 2)
;
```

Вы увидите, что там, где сравниваются два одинаковых числа, результатом является значение `Истина`. А там, где сравниваются разные числа, результатом является значение `Ложь`. К тому же оба этих значения имеют тип `Булево` (ударение на букву «у»).

Значения типа «Булево» — это значения, получаемые в результате логических операций. Значений типа `Булево` не бесконечное множество, как чисел или строк, а всего лишь два: `Истина` и `Ложь`.

Логических операций в «1С:Элементе», как и в жизни, довольно много. Все они интуитивно понятны, и нет особой надобности в том, чтобы тренироваться в их использовании. Единственное, что может вызвать трудность, — это то, какими символами они обозначаются.

Сравнение на равенство ==

5 ==5 (`Истина`), 2==5 (`Ложь`)

Сравнение на неравенство !=

2 != 5 (`Истина`), 5 != 5 (`Ложь`)

Сравнение на строгое больше >

5 > 2 (`Истина`), 2 > 5 (`Ложь`)

Сравнение на нестрогое больше >=

2 >= 2 (`Истина`), 2 >= 5 (`Ложь`)

Сравнение на строгое меньше <

2 < 5 (`Истина`), 5 < 2 (`Ложь`)

Сравнение на нестрогое меньше <=

2 <= 2 (`Истина`), 5 <= 2 (`Ложь`)

Все перечисленные в этой таблице операции называются операциями сравнения — потому что они сравнивают два значения. Причём операции **Сравнение на равенство** и **Сравнение на неравенство** можно применять к значениям любых типов. Главное, чтобы типы были одинаковыми с одной и с другой стороны операции.

А вот оставшиеся четыре операции (больше/меньше) можно применять только к тем типам, значения которых можно сравнивать. Все такие типы являются наследниками типа `Сравнимое`. Пока просто запомните это, потому что с иерархией типов вы познакомитесь позже *(103)*.

Тернарная операция

Вы уже знаете, что в основном логические операции используются в инструкциях `если`, `для` и `пока`. Однако есть одна интересная операция, которая по своей сути похожа на очень упрощённую инструкцию `если`. Это *тернарная операция* (от латинского ternarius — «тройной»).

Тернарная операция позволяет выбрать одно из двух значений в зависимости от значения логического выражения. Тернарная операция обозначается вопросительным знаком «?» и использует двоеточие «:» для разделения значений, которые нужно выбрать.

логическое_выражение `?` ***выражение_истина*** `:` ***выражение_ложь***

Например, если ваша итоговая оценка 4 или 5, то вы получаете стипендию 3 500 рублей. А если итоговая оценка ниже 4, то стипендию вы не получаете. Эту задачу можно записать следующим образом.

```
метод Скрипт()
    пер Оценка = 4
    пер Стипендия = Оценка >= 4 ? 3500 : 0
;
```

19. Задание сложное

Обычно уроки в школе начинаются в 8:30 и заканчиваются в 15:25. В среду у вас после уроков проходит дополнительное занятие, которое занимает два часа. В зависимости от того, какой сегодня день недели, нужно узнать продолжительность вашего пребывания в школе.

Для обозначения дня недели используйте новый тип `ДеньНедели`. Например, вторник будет **ДеньНедели.Вторник**.

Для указания начала и окончания занятий используйте новый тип `Время`. Он как `ДатаВремя`, только без даты. Например, два часа дня будет **Время{14:00:00}**.

Для упрощения считайте, что такие дни недели, как суббота и воскресенье, в скрипт попасть не могут, их учитывать не нужно. Решение смотрите здесь *(295)*.

Булевы операции

Общая информация

Теперь можно познакомиться с самыми интересными логическими операциями. Кроме операций сравнения есть ещё группа логических операций, которые называются *булевы операции*.

Если операции сравнения можно было выполнять в основном над числами, строками и датами, то булевы операции выполняются только со значениями типа `Булево`. То есть со значениями `Истина` и `Ложь`.

Пока что вам трудно представить, зачем это может понадобиться. Сейчас на нескольких примерах вы сразу всё поймёте.

Для этих примеров вам понадобится записывать булевы значения прямо в тексте скрипта, то есть использовать литералы. Литералы типа `Булево` выглядят просто. Значение `Истина` обозначается **Истина**, а значение `Ложь` обозначается **Ложь**.

Например, чтобы создать переменную, которая будет описывать погоду за окном, можно написать так.

```
метод  Скрипт()
    пер  СегодняСолнечно  =  Истина
;
```

Если в другой переменной вы хотите уточнить, есть на улице дождь или нет, можно написать так.

```
метод  Скрипт()
    пер  СегодняСолнечно  =  Истина
    пер  ИдетДождь  =  Ложь
;
```

Попробуйте и посмотрите в отладке, чему равны переменные.

Обратите внимание, что среда разработки «1С:Предприятие.Элемент» раскрашивает слова **Истина** и **Ложь** синим цветом. Это зарезервированные слова. Это значит, что переменную с таким именем, если вам вдруг захочется, создать нельзя.

> **Примечание:**
>
> Полный список зарезервированных слов смотрите в руководстве разработчика.

Логическое «и»

Чтобы познакомиться с первой булевой операцией, создайте две переменные: **ЯУмеюПлавать** и **РядомЕстьМоре**. Задайте им те значения, которые есть на самом деле. Здесь, для примера, они будут иметь значение `Истина`.

```
метод Скрипт()
    пер ЯУмеюПлавать = Истина
    пер РядомЕстьМоре = Истина
;
```

Теперь вам нужно с помощью логической операции узнать, будете вы плавать в море или нет.

Сначала попробуйте сказать это обычными словами. Наверное, получится что-то вроде: «Если я умею плавать и если рядом есть море, то тогда я буду плавать в море».

Теперь посмотрите, как эта же фраза выглядит на «1С:Элементе».

```
метод Скрипт()
    пер ЯУмеюПлавать = Истина
    пер РядомЕстьМоре = Истина
    пер ЯБудуПлаватьВМоре = ЯУмеюПлавать и РядомЕстьМоре
;
```

Чтобы сказать то же самое на «1С:Элементе», используется операция **логическое и**. Как она работает, вы можете проверить сами на этом примере. Запустите его в отладке, измените значения переменных.

Плавать в море вы будете только в том случае, когда оно есть рядом и вы умеете плавать. Если моря рядом нет, то плавать просто негде. Если вы не умеете плавать, то у вас тоже ничего не получится. Это очень опасно, вы можете утонуть. Ну, а если и моря нет и плавать вы не умеете, то тем более поплавать в море вам не удастся.

Операция **логическое и** возвращает значение `Истина` только в том случае, когда оба операнда имеют значение `Истина`. Во всех остальных случаях она возвращает значение `Ложь`.

Для обозначения операции **логическое и** используется русская буква «и». Эта буква строчная, поскольку это зарезервированное слово, а в «1С:Элементе» все зарезервированные слова пишутся строчными буквами.

Логическое «или»

Теперь решите другую задачу. Создайте две переменные **УМеняЕстьЛодка** и **УМеняЕстьПлот**. Можете задать им любые значения. Здесь, для примера, они будут иметь значение `Истина`.

```
метод Скрипт()
    пер УМеняЕстьЛодка = Истина
    пер УМеняЕстьПлот = Истина
;
```

Вам нужно с помощью логической операции узнать, сможете вы переплыть через озеро или нет.

Сначала попробуйте сказать это обычными словами. Наверное, получится что-то вроде: «Если у меня есть плот или лодка, то я могу переплыть через озеро».

А теперь посмотрите, как эта же фраза выглядит на «1С:Элементе».

```
метод Скрипт()
    пер УМеняЕстьЛодка = Истина
    пер УМеняЕстьПлот = Истина
    пер ЯМогуПереплытьЧерезОзеро = УМеняЕстьЛодка или УМеняЕстьПлот
;
```

Чтобы сказать то же самое на «1С:Элементе», используется операция **логическое или**. Как она работает, вы можете проверить сами на этом примере, изменив значения переменных.

Если у вас есть лодка, вы можете переплыть через озеро? Да.

Если у вас не лодка, а плот, тогда сможете? Да, сможете.

А если у вас есть и лодка, и плот? Конечно! Даже два раза! Один раз на лодке, а другой — на плоту (шутка)!

А в каком случае вам не удастся переплыть через озеро? Правильно. Когда у вас нет ни лодки, ни плота.

Операция **логическое или** работает аналогично операции **логическое и**, только в отношении значения `Ложь`. Она возвращает `Ложь` только в том случае, когда оба операнда имеют значение `Ложь`. Во всех остальных случаях она возвращает значение `Истина`.

Совмещение операций

Усложните пример и познакомьтесь с тем, что в одном выражении могут использоваться сразу несколько булевых операций.

Создайте три переменные: **ЯУмеюПлавать**, **РядомЕстьМоре** и **РядомЕстьБассейн**. Задайте им те значения, которые есть на самом деле. Здесь, для примера, они снова будут иметь значение **Истина**.

```
метод Скрипт()
    пер ЯУмеюПлавать = Истина
    пер РядомЕстьМоре = Истина
    пер РядомЕстьБассейн = Истина
;
```

Вопрос, решением которого вы займётесь, будет практически тем же самым. Чему будет равна переменная **ЯБудуПлавать**?

Если вы будете отвечать словами, то скажете: «Я буду плавать, если я умею плавать и рядом есть море или бассейн». Попробуйте записать это на «1С:Элементе».

```
метод Скрипт()
    пер ЯУмеюПлавать = Истина
    пер РядомЕстьМоре = Истина
    пер РядомЕстьБассейн = Истина
    пер ЯБудуПлавать = ЯУмеюПлавать и РядомЕстьМоре или РядомЕстьБассейн
;
```

На первый взгляд кажется, что всё хорошо. Но нужно проверить.

Например, вы не умеете плавать. В этом случае неважно, есть рядом бассейн или нет, плавать вы не должны. Но что говорит вам скрипт? Попробуйте.

```
метод Скрипт()
    пер ЯУмеюПлавать = Ложь
    пер РядомЕстьМоре = Истина
    пер РядомЕстьБассейн = Истина
    пер ЯБудуПлавать = ЯУмеюПлавать и РядомЕстьМоре или РядомЕстьБассейн
;
```

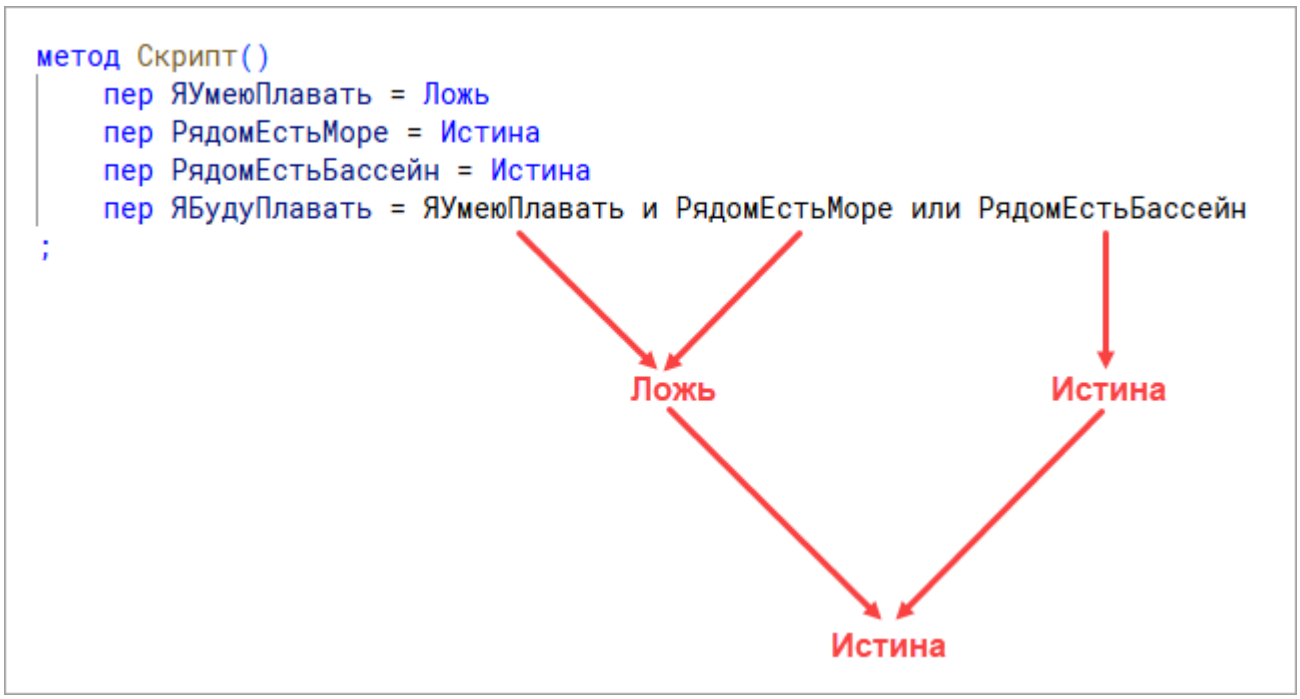
Скрипт говорит вам, что плавать вы будете. Получается какое-то «опасное» программирование. С таким программированием и утонуть недолго!

В чём же дело? Может быть, вы написали неправильные операции? Нет, правильные.

Просто вы не учли, что если несколько булевых операций встречаются в одном выражении, то у них есть определённый порядок выполнения. Точно так же, как было с арифметическими операциями «+», «-», «*» и «/».

Сначала выполняется операция **логическое и**, а затем уже выполняется операция **логическое или**. Если подряд встречаются несколько одинаковых операций, то они выполняются в той последовательности, в которой написаны.

Поэтому в примере выражение было посчитано таким образом.



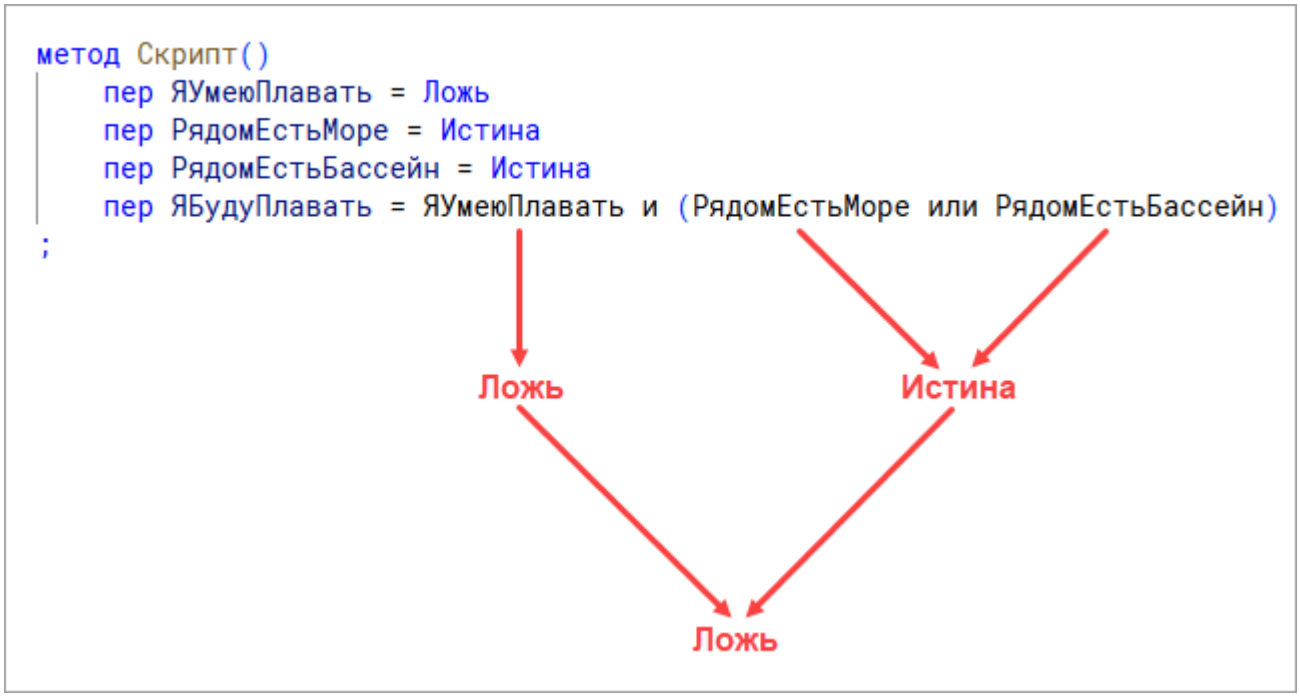
Сначала была выполнена операция **логическое и**, которая дала `Ложь`. Затем операция **логическое или**, результатом которой явилась `Истина`.

На самом деле для вас в этом выражении важны две вещи: что вы умеете плавать и что есть где плавать. Море это или бассейн — не так важно.

Поэтому правильным решением будет **РядомЕстьМоре или РядомЕстьБассейн** заключить в скобки. Попробуйте.

```
метод Скрипт()
    пер ЯУмеюПлавать = Ложь
    пер РядомЕстьМоре = Истина
    пер РядомЕстьБассейн = Истина
    пер ЯБудуПлавать = ЯУмеюПлавать и (РядомЕстьМоре или РядомЕстьБассейн)
;
```

Тогда «1С:Элемент» сначала вычислит то выражение, которое в скобках. То есть узнает, есть где плавать или плавать негде. Потом уже он поинтересуется вашим умением плавать.



Чтобы закрепить этот пример, измените его условие. Создайте три переменные: **ЯУмеюПлавать**, **РядомЕстьМоре** и **РядомЕстьВанна**. Задайте им те значения, которые были в последнем примере: `Ложь`, `Истина` и `Истина`.

```
метод Скрипт()
    пер ЯУмеюПлавать = Ложь
    пер РядомЕстьМоре = Истина
    пер РядомЕстьВанна = Истина
;
```

Вычислить нужно переменную, которая называется **ЯБудуКупаться**. Будьте внимательны: имя переменной изменилось не просто так.

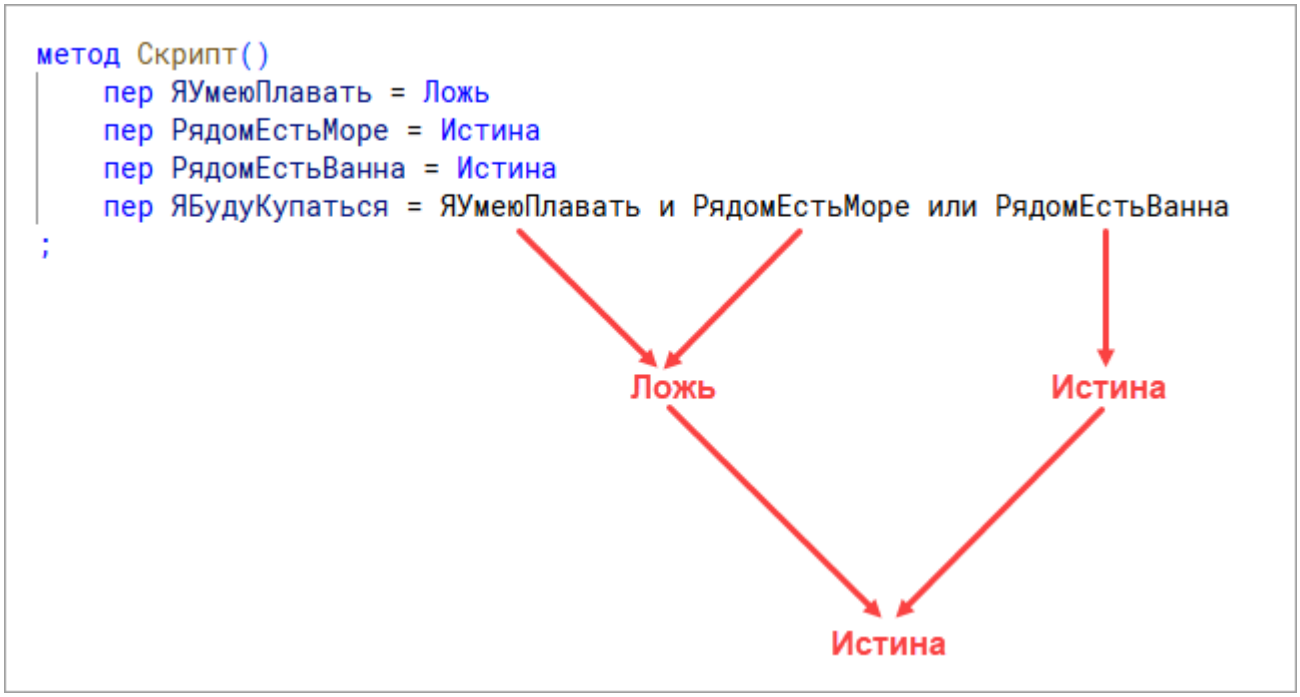
Попробуйте записать выражение на «1С:Элементе» и потом сравните с тем, что ниже.

```
метод Скрипт()
    пер ЯУмеюПлавать = Ложь
    пер РядомЕстьМоре = Истина
    пер РядомЕстьВанна = Истина
```

```
    пер ЯБудуКупаться = ЯУмеюПлавать и РядомЕстьМоре или РядомЕстьВанна
;
```

Запустите скрипт в режиме отладки и посмотрите, какой получается результат, если изменить значения переменных.

Почему в этом случае скобки не понадобились, хотя пример внешне очень похож на предыдущий? Скобки не понадобились потому, что «естественный» порядок выполнения логических операций в этом выражении вполне вас устраивает.



Действительно, умение плавать важно только тогда, когда вы собираетесь зайти в море. Чтобы искупаться в ванне, умение плавать вам не нужно.

Поэтому правильно, что сначала выполняется операция **логическое и** — выясняется, можете ли вы купаться в море. Затем уже проверяется наличие ванны, для использования которой способность плавать не требуется.

Логическое «не»

В заключение нужно сказать ещё об одной булевой операции — операции **логическое не**. Она очень простая. Она возвращает булево значение, противоположное тому, которое имеется.

Чтобы посмотреть, как она работает, вы можете использовать такой пример.

```
метод Скрипт()
    пер ПасмурнаяПогода = Истина
    пер СолнечнаяПогода = не ПасмурнаяПогода
;
```

Обычно с использованием этой операции никаких трудностей не возникает. Единственное, что может вам понадобиться, это порядок выполнения булевых операций. Полностью он выглядит так:

- сначала выполняется то, что в скобках;
- затем операция **логическое не**;
- затем операция **логическое и**;
- затем операция **логическое или**.

20. Задание простое

Чтобы по пути на работу защититься от дождя, вы берёте с собой зонт. С помощью переменных **ЯИдуНаРаботу** и **ИдетДождь** вычислите значение переменной **НадоВзятьЗонт**. Проверьте, что ваша инструкция правильно работает при любых значениях исходных переменных. Решение смотрите здесь *(295)*.

21. Задание простое

В каких случаях вы не ходите в школу? Когда выходной (праздники, каникулы) и когда вы болеете. С помощью переменных **СегодняВыходной** и **ЯБолею** вычислите значение переменной **ЯНеИдуВШколу**. Проверьте, что ваша инструкция правильно работает при любых значениях исходных переменных. Решение смотрите здесь *(295)*.

22. Задание сложное

В хорошую погоду, когда у вас нет занятий, вы всегда идёте гулять. С помощью переменных **ХорошаяПогода**, **СегодняВыходной** и **СегодняПраздник** вычислите значение переменной **ЯИдуГулять**. Проверьте, что ваша инструкция правильно работает при любых значениях исходных переменных. Решение смотрите здесь *(296)*.

23. Задание сложное

Вы работаете только по будним дням. С помощью переменных **СегодняСуббота** и **СегодняВоскресенье** вычислите значение переменной **ЯИдуНаРаботу**. Проверьте, что ваша инструкция правильно работает при любых значениях исходных переменных. Решение смотрите здесь *(296)*.

Инструкция «если»

Общая информация

Настало время познакомиться с ещё одной инструкцией, которая используется в «1С:Элементе».

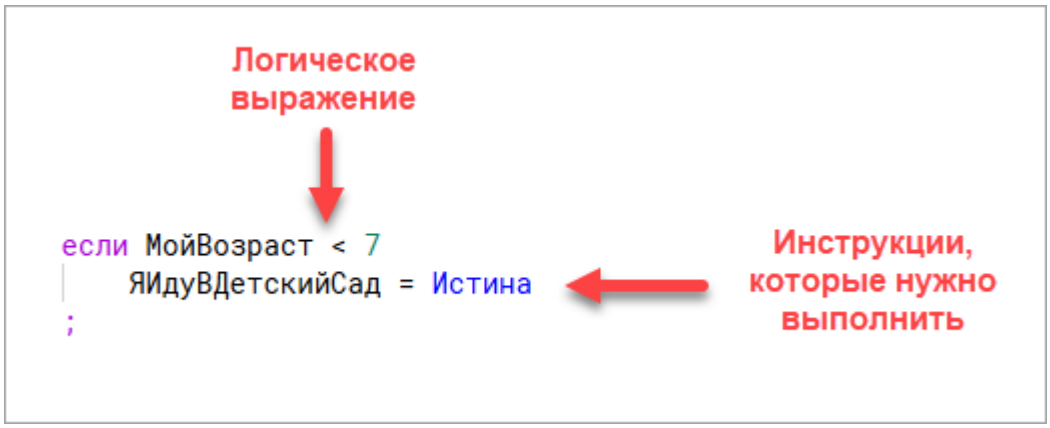
Если бы вы использовали только инструкцию присваивания, то ваш скрипт выглядел бы как прямая дорога из пункта «А» в пункт «Б». От начала к завершению. Так происходит потому, что инструкции присваивания выполняются одна за другой в том порядке, в котором они написаны.

На самом деле большинство программ выглядят по-другому. Между началом программы и её завершением существует много разных путей, по которым может пойти исполнение программы. Для этого есть много разных причин. Например, это зависит от того, какие данные уже есть в программе. Если есть все необходимые данные, то исполнение пойдёт по одному пути. Если каких-то данных не хватает, то исполнение пойдёт по другому пути.

Также это зависит от того, что вы хотите получить в результате. Если, например, из своего электронного дневника вы хотите узнать, будет завтра урок математики или нет, то это будет один путь — простой. Если вы хотите получить расписание уроков на всю неделю с указанием кабинетов, учителей и того, что задано, то это будет другой путь — сложный. Скрипту нужно будет обойти много разных мест и собрать для вас всю информацию, которую вы просите.

Инструкция «если»

Для того чтобы направить исполнение скрипта по одному или по другому пути, существует инструкция `если`. Выглядит она очень просто.




```
метод Скрипт()
    пер МойВозраст = 17
    пер ЯИдуВДетскийСад: Булево
    если МойВозраст < 7
        ЯИдуВДетскийСад = Истина
    ;
;
```

Инструкция `если` является *составной*. Она позволяет выделить группу других инструкций, которые будут выполняться не всегда, а только тогда, когда в результате вычисления логического выражения получается `Истина`.

В примере вычисляется логическое выражение **МойВозраст < 7**. Группа других инструкций представлена одной инструкцией пер **ЯИдуВДетскийСад = Истина**, но их может быть любое количество.

Слово **если** и точка с запятой «;» являются обязательными в этой инструкции. Логическое выражение пишется после слова **если**. Точка с запятой «;» показывает, где заканчиваются инструкции, выполнение которых зависит от условия.

Чтобы посмотреть, как это работает, скопируйте пример в свой скрипт, установите точку останова на строке **если МойВозраст < 7**, запустите скрипт в режиме отладки и посмотрите значение выражения **МойВозраст < 7**.

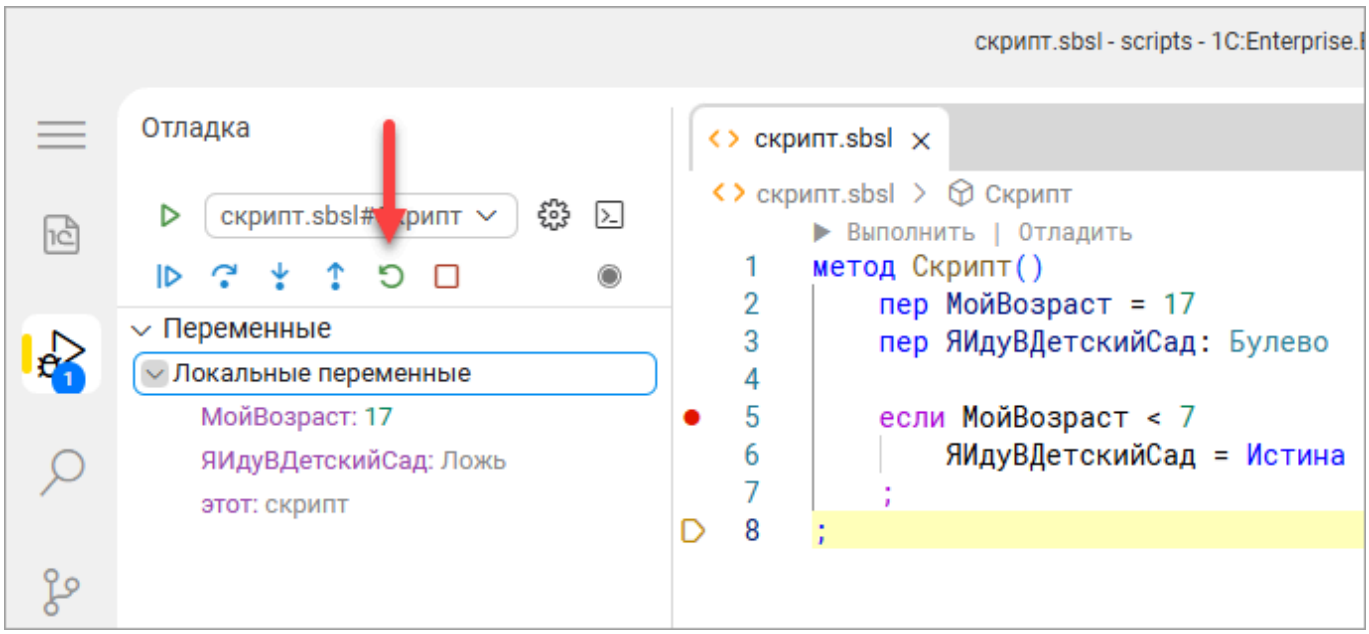
Вы увидите, что оно равно `Ложь`. И это правильно. Ваш возраст больше, чем 7 лет.

Раз выражение ложно, значит, «1С:Элемент» должен пропустить все инструкции, которые написаны внутри инструкции `если`. Проверьте это с помощью пошагового исполнения (**F11**).

Сделайте один шаг, и вы увидите, что «1С:Элемент» перейдёт на точку с запятой «;», которая является концом скрипта. При этом переменная **ЯИдуВДетскийСад** сохранит своё значение по умолчанию, `Ложь`, потому что инструкция её изменения не выполнялась.

Теперь познакомьтесь с тем, как поведёт себя «1С:Элемент» в том случае, когда логическое выражение истинно. Для этого вы не будете изменять текст скрипта. Вы воспользуетесь возможностью изменения значений переменных прямо в процессе отладки. Чтобы это сделать, вам потребуется сначала перезапустить отладку.

Для этого выполните команду **Перезапустить**. Она находится в представлении **Отладка** в командной панели — пятая по счёту.



Это же действие можно выполнить из главного меню: **Главное меню > Запустить > Перезапустить**.

Среда разработки «1С:Предприятие.Элемент» спросит: **"Run 1C Script Executor" уже выполняется. Вы хотите запустить другой экземпляр?**. Нажмите **ОК**.

Среда разработки запустит новый экземпляр отладки и снова остановится на той же точке останова, на строке **если МойВозраст < 7**.

Теперь, не изменяя текст скрипта, вы можете просто изменить значение имеющихся переменных. Это удобно, чтобы посмотреть, как поведёт себя скрипт при других значениях.

Откройте раздел **Локальные переменные**. Дважды щёлкните мышью на переменной **МойВозраст**. Введите **6** и нажмите **ОК**.

Сделайте один шаг. Исполнение зайдёт внутрь инструкции `если`.

Теперь, если сделаете ещё пару шагов, вы дойдёте до конца скрипта. А значение переменной **ЯИдуВДетскийСад** будет равно `Истина`.

Остановите отладку.

Ключевое слово «иначе»

До сих пор вы рассматривали самую простую форму инструкции `если`. На деле она может быть более сложной.

В вашем примере «особенные» действия выполняются только в том случае, когда ваш возраст меньше 7 лет. Представьте, что вам нужно принять решение из двух вариантов. Если меньше 7 лет, вы идёте в детский сад. Во всех других случаях вы идёте в школу. Как это записать на «1С:Элементе»?

Оказывается, очень просто. Для этого используется ключевое слово `иначе`.

```
метод Скрипт()
    пер МойВозраст = 17
    пер ЯИдуВДетскийСад: Булево
    если МойВозраст < 7
        ЯИдуВДетскийСад = Истина
    иначе

    ;
;
```

Доработайте пример так, чтобы в результате его работы у вас изменялись две переменные: **ЯИдуВДетскийСад** и **ЯИдуВШколу**. Чтобы они принимали значение `Истина` в зависимости от возраста, указанного в переменной **МойВозраст**.

Сравните свой результат с тем, что показано ниже.

```
метод Скрипт()
    пер МойВозраст = 17
    пер ЯИдуВДетскийСад: Булево
    пер ЯИдуВШколу: Булево
    если МойВозраст < 7
        ЯИдуВДетскийСад = Истина
    иначе
        ЯИдуВШколу = Истина
    ;
;
```

Запустите скрипт в режиме отладки и проверьте по шагам правильность работы примера для разных значений возраста. Изменяйте возраст прямо в процессе отладки.

Блок инструкций «иначе если»

Часто в инструкции `если` анализируется не одно, а несколько условий. Например, в одно прекрасное утро вы проснулись и вспомнили, что вам нужно идти учиться. Но вы не помните, куда именно: в детский сад, в школу или в институт.

Зато вам известны правила. До 7 лет нужно идти в детский сад. После детского сада нужно идти в школу. Учиться в школе заканчивают в 18 лет. А, например, в 19 лет поступают в институт. Чтобы записать этот алгоритм на «1С:Элементе», вам понадобится ещё одна фраза из ключевых слов — `иначе если`.

Напишите эту фразу вместо ключевого слова `иначе`. Затем напишите условие, при котором вам нужно идти в школу.

```
метод Скрипт()
    пер МойВозраст = 17
    пер ЯИдуВДетскийСад: Булево
    пер ЯИдуВШколу: Булево
    если МойВозраст < 7
        ЯИдуВДетскийСад = Истина
    иначе если МойВозраст >= 7 и МойВозраст < 19
        ЯИдуВШколу = Истина
    ;
;
```

В конце примера снова добавьте слово `иначе` и укажите, что во всех остальных случаях вы идёте в институт. Попробуйте сделать пример самостоятельно.

Чтобы проверить свой результат, сравните его с тем, что ниже.

```
метод Скрипт()
    пер МойВозраст = 17
    пер ЯИдуВДетскийСад: Булево
    пер ЯИдуВШколу: Булево
    пер ЯИдуВИнститут: Булево
    если МойВозраст < 7
        ЯИдуВДетскийСад = Истина
    иначе если МойВозраст >= 7 и МойВозраст < 19
        ЯИдуВШколу = Истина
    иначе
        ЯИдуВИнститут = Истина
    ;
;
```

В режиме отладки пройдите по разным веткам инструкции `если` и посмотрите, как она работает.

Вы увидите, что выражения, которые указаны в инструкции `если`, вычисляются и анализируются по очереди — в том порядке, в котором они написаны. Как только попадается истинное выражение, выполняется соответствующая ветка инструкции `если`. После этого исполнение переходит на инструкцию, следующую за `если`, в данном случае — на точку с запятой «;», закрывающую скрипт. То есть инструкции, расположенные в тех ветках, где выражение было ложно, или в тех ветках, до которых проверка выражений не дошла, просто не выполняются.

24. Задание простое

Вы с другом каждый день играете в игру через Интернет. В будний день родители разрешают вам играть только один час. В выходные дни вы можете играть по 4 часа. Используя переменную **НомерДняНедели**, вычислите, сколько часов вы можете играть с другом в выбранный день. Результат поместите в переменную **ВремяДляИгры**. Решение смотрите здесь *(296)*.

25. Задание простое

В задании 24 нужно учесть, что номер дня недели может быть задан с ошибкой. В случае ошибки ваш скрипт должен в переменную **Сообщение** записать текст ошибки. Решение смотрите здесь *(296)*.

26. Задание сложное

Супермаркет работает с 9 часов утра до 8 часов вечера. Вам нужно сходить в супермаркет и купить кефир. Если кефира не будет, то ряженку. Если не будет ни того ни другого, тогда нужно зайти в круглосуточный магазин и купить молоко, если оно там есть. Используйте переменные **ЕстьКефир**, **ЕстьРяженка**, **ЕстьМолоко** и **ТекущийЧас**. Название своей покупки в виде строки поместите в переменную **МояПокупка**. Решение смотрите здесь *(297)*.

Красивый скрипт

Синтаксический отступ

Вы уже знаете, что ваш скрипт должен быть красивым. Вы помните, что имена переменных должны быть удобными, понятными. Помните, что выражения должны быть записаны понятно и «читабельно». Теперь настал удобный момент рассказать о том, как должен выглядеть текст вашего скрипта.

Вы, наверное, уже привыкли к тому, что среда разработки «1С:Предприятие.Элемент» сама раскрашивает слова в вашем скрипте, это удобно. Может быть, вы обратили внимание на то, что среда разработки сама делает отступ, когда вы пишете инструкцию `если`. Если вы этого не заметили, то поставьте небольшой эксперимент. В скрипте напишите:

```
метод Скрипт()
    пер Тест: Число
    если Тест == 2
;
```

Убедитесь, что курсор стоит после цифры 2, и нажмите **Ввод**.

Вы увидите, что на новой строке курсор встанет не под буквой **е**, а с некоторым сдвигом вправо. Этот сдвиг называется *синтаксическим отступом*. Он помогает лучше и легче читать текст скрипта. Когда он используется, подчинённые инструкции, расположенные внутри одной ветки, оказываются выделены визуально.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"
    если ТекущийЧас >= 9 и ТекущийЧас < 20
        если ЕстьКефир
            МояПокупка = "Кефир"
        иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
```

```
    ;
;
```

Посмотрите, какая «каша» получилась бы, если бы синтаксического отступа не было.

```
метод Скрипт()
пер ЕстьКефир = Истина
пер ЕстьРяженка = Истина
пер ЕстьМолоко = Истина
пер ТекущийЧас = ДатаВремя.Сейчас().Час
пер МояПокупка = "Ничего"
если ТекущийЧас >= 9 и ТекущийЧас < 20
если ЕстьКефир
МояПокупка = "Кефир"
иначе если ЕстьРяженка
МояПокупка = "Ряженка"
;
;
если МояПокупка == "Ничего" и ЕстьМолоко
МояПокупка = "Молоко"
;
;
```

Прочитать такую «кашу» и разобраться в ней очень сложно.

По этой причине синтаксический отступ — важная, обязательная часть скрипта. Именно поэтому среда разработки делает её автоматически, когда вы набираете текст инструкции последовательно.

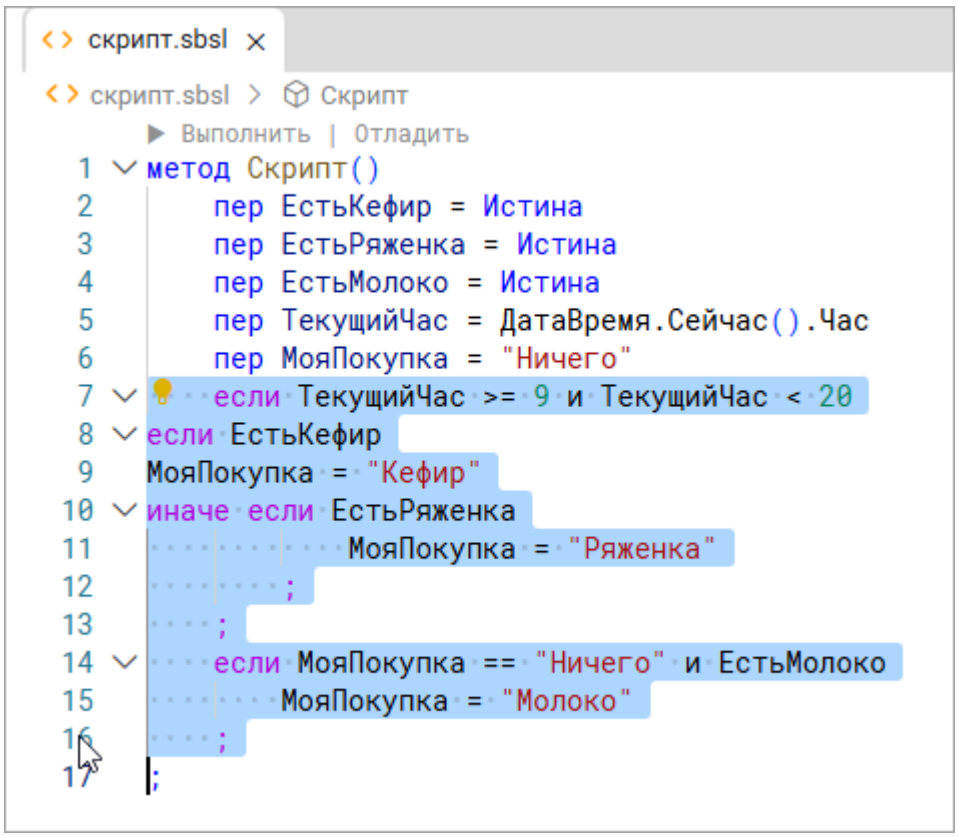
Однако далеко не всегда вы работаете именно так. Даже при выполнении простых примеров вы видите, что приходится изменять то, что уже написано. Вставлять новые строки. Копировать из одного места в другое. В этих случаях среда разработки не может автоматически сделать синтаксический отступ. В результате у вас вполне может получиться такой текст.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"
    если ТекущийЧас >= 9 и ТекущийЧас < 20
если ЕстьКефир
МояПокупка = "Кефир"
иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
    ;
;
```

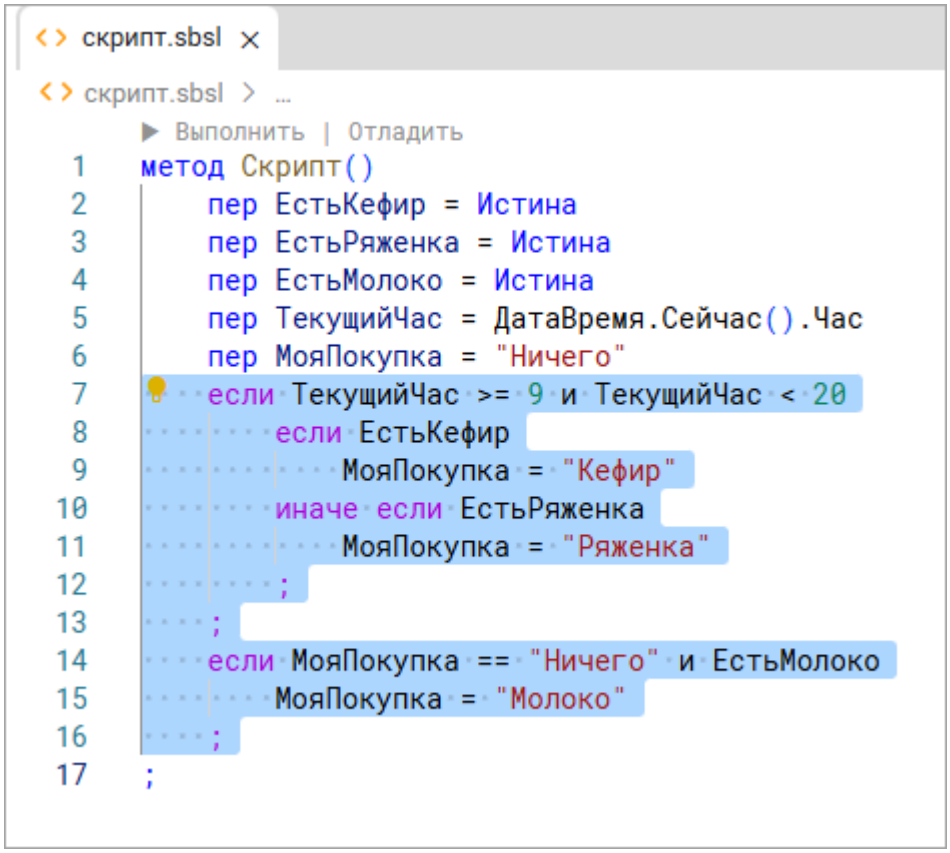
Как с ним быть? Двигать каждую строку вручную, чтобы было «красиво»? Это скучно, долго и неинтересно.

В среде разработки есть команда, которая может сделать это автоматически. Пользоваться этой командой очень просто. Сначала вы выделяете текст, который нужно отформатировать. Только выделять лучше не ровно нужные строки, а чуть больше сначала и в конце. Чтобы было понятно расположение остального текста.

Чтобы выделить, нажмите мышью на номере той строки, начиная с которой нужно выделить, и, не отпуская кнопку мыши, ведите её до последней строки, которую нужно выделить. Отпустите кнопку мыши.



Теперь на выделенном фрагменте вызовите контекстное меню и нажмите **Форматировать выделенный фрагмент**.



Если форматировать нужно почти весь текст, можете ничего не выделять, а просто нажмите **Форматировать документ** в контекстном меню области редактирования.

Пустые строки

Помимо синтаксического отступа есть и другие способы сделать текст более понятным. Например, один из хороших способов — это добавление пустых строк в текст скрипта. Они позволяют разделить текст на несколько отдельных смысловых блоков.

Если взять пример с инструкцией `если`, то такие смысловые блоки напрашиваются сами собой. Каждая ветка этой инструкции является отдельным смысловым блоком. Поэтому, если перед началом ветки вы добавите пустую строку, это только улучшит читаемость вашего скрипта.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"

    если ТекущийЧас >= 9 и ТекущийЧас < 20
        если ЕстьКефир
            МояПокупка = "Кефир"

        иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;
```

```
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
    ;
;
```

Комментарии

Другой хороший и очень часто используемый приём — создание комментариев. *Комментарий* — это пояснение к скрипту, которое находится прямо в тексте скрипта.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"

    // Супермаркет.
    если ТекущийЧас >= 9 и ТекущийЧас < 20
        если ЕстьКефир
            МояПокупка = "Кефир"

        иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;

    // Круглосуточный магазин.
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
    ;
;
```

Комментарии всегда начинаются с двух косых черт «//». Среда разработки выделяет комментарии зелёным цветом.

Комментарии, конечно, могут быть разными. Сформулировать правило: «Комментарии должны быть вот такими...» — невозможно. Ситуации бывают разные.

В большинстве случаев нужно стремиться к тому, чтобы описать не ЧТО делается, а чтобы пояснить, ЗАЧЕМ это делается.

Главная сложность тут заключается в том, что в тот момент, когда вы пишете скрипт, вы прекрасно понимаете, зачем вы делаете то или это. Вам кажется, что писать по этому поводу комментарии совершенно не нужно. Ведь и так всё понятно!

Но проходит неделя, месяц, год… Вы открываете свой старый скрипт и вдруг с удивлением обнаруживаете, что в нём ничего не понятно. Иногда даже бывает такое ощущение, что всё это писали не вы, а кто-то другой.

Поэтому всегда нужно самого себя немного заставлять писать комментарии. В этот момент можно представить, что вы пишете скрипт не для себя, а для другого программиста. Другому программисту нужно будет разобраться в нём без вашей помощи.

Если вы себе это представите, то сразу поймёте, например, что такой комментарий будет бесполезен.

```
// Объявить переменные.
пер ЯИдуВДетскийСад: Булево
пер ЯИдуВШколу: Булево
пер ЯИдуВИнститут: Булево
```

Однако, если вы напишете комментарий так, то он, наоборот, очень поможет и вам, и другим.

```
// Создать переменные с начальным значением Ложь,
// чтобы потом установить в Истина только ту,
// которая соответствует возрасту.
пер ЯИдуВДетскийСад: Булево
пер ЯИдуВШколу: Булево
пер ЯИдуВИнститут: Булево
```

Умение писать нужные и полезные комментарии можно приобрести только с опытом. Поэтому пробуйте, пишите, не ленитесь.

Чтобы комментарии выглядели понятно и красиво, придерживайтесь нескольких простых правил.

Однострочные комментарии

При написании скриптов фирма «1С» рекомендует не писать строки длиннее 120 символов. Если ваш комментарий примерно помещается в 120 символов, можно оформить его как однострочный комментарий.

```
пер МойВозраст = 17

// Присвоить переменным начальные значения.
пер ЯИдуВДетскийСад: Булево
пер ЯИдуВШколу: Булево
пер ЯИдуВИнститут: Булево
```

Однострочные комментарии располагаются на отдельной строке и начинаются с двух косых черт «//». Всё, что следует дальше до конца строки, считается комментарием.

Текст комментария соответствует правилам русского языка: начинается с прописной буквы, предложения заканчиваются точкой. Между «//» и текстом комментария оставляйте пробел.

Поскольку инструкции, следующие за комментарием, представляют собой некоторую группу, требующую пояснений (именно поэтому вы и пишете комментарий), вставляйте перед однострочным комментарием пустую строку. Она дополнительно выделит этот смысловой блок.

> **Примечание:**
>
> Подробнее о рекомендациях по написанию кода читайте здесь *(228)*.

Многострочные комментарии

Если ваш комментарий длиннее 120 символов или состоит из нескольких предложений, которые хочется начать с новой строки, используйте многострочные комментарии.

```
пер МойВозраст = 17

/* Присвоить переменным начальные значения,
   чтобы потом установить в Истина только ту переменную,
   которая соответствует возрасту */
пер ЯИдуВДетскийСад: Булево
пер ЯИдуВШколу: Булево
пер ЯИдуВИнститут: Булево
```

Многострочный комментарий начинается с косой черты и звёздочки «/*» и продолжается до тех пор, пока не встретится звёздочка и косая черта «*/». В остальном правила написания многострочных комментариев такие же, как и однострочных. Вторую и следующие строки комментария обычно выравнивают по началу текста в первой строке комментария.

Многострочные комментарии, помимо пояснения текста скрипта, имеют и другое назначение. Порой нужно временно исключить из выполнения какие-то строки скрипта. Удалять их совсем нельзя, они нужны или могут понадобиться в будущем. Просто в данный момент вам нужно, чтобы скрипт выполнился без этих строк.

В такой ситуации вы можете закомментировать эти строки, заключив их в многострочный комментарий. Почему именно в многострочный? Потому что вам потребуется вставить только символ начала комментария и символ его окончания. Если бы вы использовали для этого однострочный комментарий, вам пришлось бы комментировать каждую строку отдельно.

В следующем примере закомментирована последняя инструкция `если`.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"

    // Супермаркет.
    если ТекущийЧас >= 9 и ТекущийЧас < 20
        если ЕстьКефир
            МояПокупка = "Кефир"
        иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;
/*
    // Круглосуточный магазин.
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
    ;*/
;
```

Короткие однострочные комментарии

Чтобы дать небольшие пояснения к отдельным строкам скрипта, пишите их в виде однострочных комментариев на той же строке, что и сама инструкция.

В этом случае обычно комментарий начинается со строчной буквы и точка в конце не ставится.

```
если МойВозраст < 7    // ясли не учитываем
    ЯИдуВДетскийСад = Истина

иначе если МойВозраст >= 7 и МойВозраст < 19
    ЯИдуВШколу = Истина

иначе    // считаем, что в институт можно пойти в любом возрасте
    ЯИдуВИнститут = Истина
;
```

Инструкция «для по»

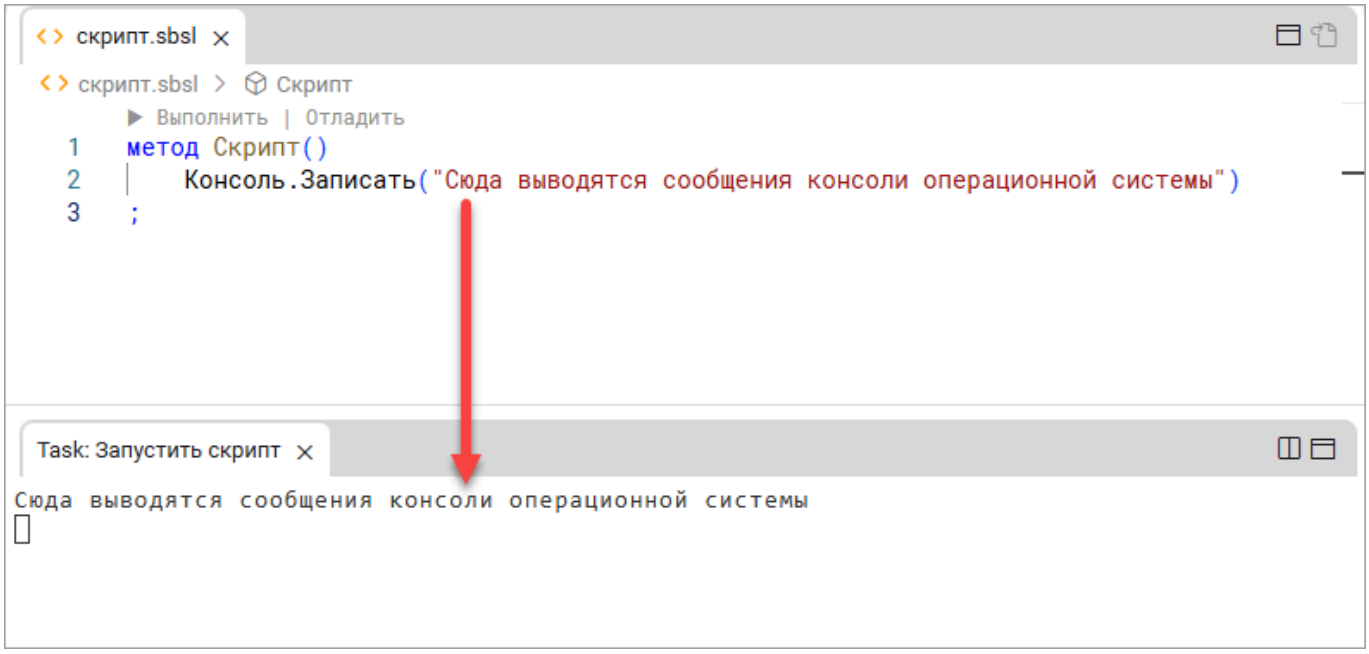
Постановка задачи

Представьте себе такую задачу. Вам нужно напечатать список. В этом списке должно быть указано, сколько занятий у вас в понедельник, во вторник и так далее:

- в 1-й день — 6 занятий;
- во 2-й день — 8 занятий;
- в 3-й день — 7 занятий и т. д.

Чтобы это сделать, у вас есть две переменные. Одна — **ДеньНедели**. В ней вы можете хранить порядковый номер дня недели. Другая — **Занятий**. В ней вы можете хранить количество занятий.

Печатать вы не умеете. Поэтому упростите задачу. Вместо того чтобы печатать, выводите сообщения в терминал. Когда вы будете запускать свой скрипт на исполнение в среде разработки, вы сможете увидеть эти сообщения в представлении **Терминал**.



Текст, который вам нужно будет вывести, будет формироваться следующим образом:

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6
    Консоль.Записать("В " + ДеньНедели.Представление() + "-й день " + Занятий.Представление() + " занятий.")
;
```

В результате в терминал будет выведено значение **"В 1-й день 6 занятий."**.

Эта строка «склеивается» из нескольких частей, из нескольких строк. В ней используются переменные **ДеньНедели** и **Занятий**.

Поскольку эти переменные содержат числа, в «голом» виде их в этой инструкции использовать нехорошо. Сначала нужно сделать из них строки. Для этого два раза используется метод `Число.Представление()`. Он преобразует числовое значение к типу `Строка` привычным для человека образом.

> **Примечание:**
>
> Примеры использования метода `Число.Представление()` смотрите здесь *(159)*.

Неявное преобразование типа

Небольшое отступление.

Несмотря на то что «1С:Элемент» использует статическую типизацию, несмотря на то что тип любой переменной всегда определён в момент её описания, несмотря на то что операция `+` обычно выполняется над значениями одинакового или похожего типа, для типа `Строка` здесь сделано послабление.

Если в операции `+` первым операндом является значение типа `Строка`, то остальные операнды «1С:Элемент» сам автоматически приведёт к типу `Строка`. То есть вам не нужно самостоятельно выполнять такое преобразование.

Поэтому вы можете написать проще, это будет читаться лучше и выполняться эффективнее.

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий.")
;
```

Однако всё не так радужно, как хотелось бы.

Для того чтобы преобразовать значение некоторого типа к типу `Строка`, «1С:Элемент» будет использовать метод `Объект.ВСтроку()`. Этот метод наследуют почти все типы «1С:Элемента». Иерархию типов *(103)* вы будете изучать позже, сейчас достаточно запомнить, что метод с именем **ВСтроку()** есть почти у всех типов.

Однако есть и другой похожий метод — `Объект.Представление()`. Он тоже преобразует значение к типу `Строка`, но привычным для человека образом.

Чтобы вам было понятно, в чём может заключаться разница между этими двумя методами, сделайте такой пример.

```
метод Скрипт()
    пер Тест = 123456789.345
    Консоль.Записать(Тест.ВСтроку())
    Консоль.Записать(Тест.Представление())
;
```

Вы получите следующий результат.

```
123456789.345
123 456 789,345
```

Результат, выдаваемый методом `Объект.ВСтроку()`, больше похож на литерал значения, а результат метода `Объект.Представление()` больше похож на привычное представление чисел.

Отсюда понятно, что если вам нужно сформировать любое понятное для человека сообщение, то можно использовать неявное преобразование типа при конкатенации. Также его можно использовать в том случае, когда оба метода дают одинаковый результат. Например, при выводе целых чисел.

Если же вы выводите значения, представления которых сильно отличаются от литерала, лучше самостоятельно преобразовывать значения перед конкатенацией с помощью метода `Объект.Представление()`, тем более что некоторые унаследованные методы **Представление()** имеют перегрузки *(212)*, позволяющие указывать форматную строку.

В этом примере вы будете выводить целые числа, вас вполне устроит преобразование `Объект.ВСтроку()`. Поэтому ваш скрипт будет выглядеть так.

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий." )
;
```

Описание форматной строки можно посмотреть в справке по стандартной библиотеке, например:

- Число.Представление();
- ДатаВремя.Представление() и т. д.

Линейное исполнение кода

Итак, есть две переменные и есть инструкция. Нужно с их помощью вывести в терминал список.

Если бы вы знали только инструкцию присваивания, вы бы стали делать эту задачу так.

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")

    ДеньНедели = 2
```

```
    Занятий = 8
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")

    ДеньНедели = 3
    Занятий = 7
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
;
```

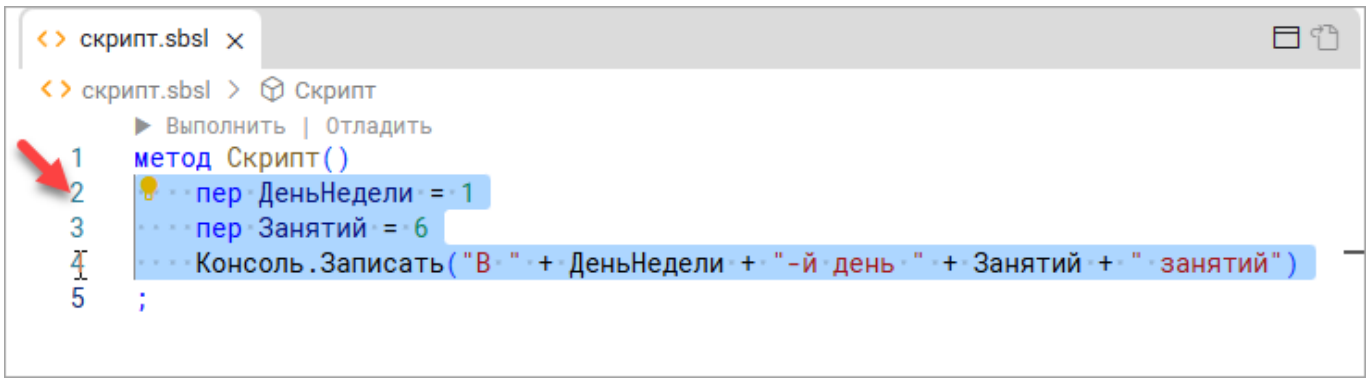
В результате в терминал будут выведены следующие строки.

```
В 1-й день 6 занятий
В 2-й день 8 занятий
В 3-й день 7 занятий
```

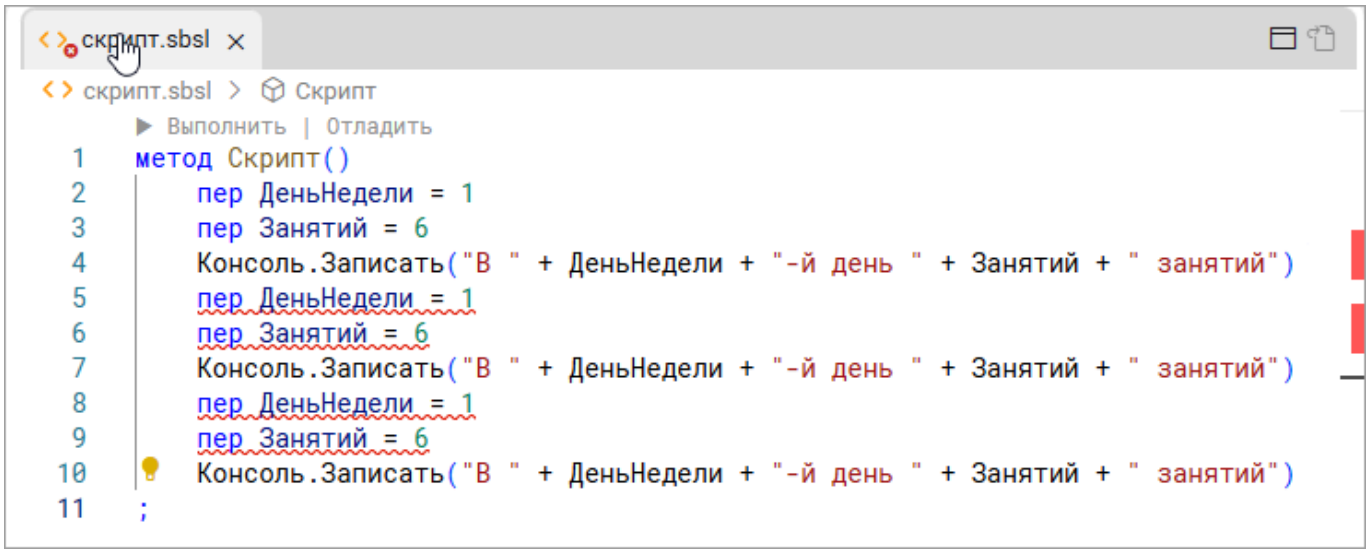
Копирование — вставка

Не поленитесь, сделайте в своей конфигурации этот пример. Чтобы три раза не писать одно и то же, напишите только три строки для первого дня. А затем скопируйте эти строки два раза, как описано далее.

Копировать и вставлять строки очень просто. Сначала выделите строки. Для этого на колонке с номерами строк нажмите на номер первой строки и ведите мышью вниз. Отпустите, когда все нужные строки будут выделены.



Затем нажимаете **Ctrl+C** и три раза **Ctrl+V**. Строки скопированы и вставлены. Осталось лишь немного их отредактировать.



Добавьте пустые строки перед каждым днём. В скопированных строках удалите **пер**, измените день недели и количество занятий. Переменные вы создали в начале и теперь только меняете значения в них.

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")

    ДеньНедели = 2
    Занятий = 8
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")

    ДеньНедели = 3
    Занятий = 7
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
;
```

Ветвление кода

Почему вы не стали доделывать этот пример до конца? Потому что каждый день делается одно и то же. Зная порядковый день недели, вы выясняете, сколько занятий в этот день. А для этого очень хорошо подходит инструкция если.

```
если ДеньНедели == 1
    Занятий = 6

иначе если ДеньНедели == 2
    Занятий = 8

// И так далее.
;
```

Расписание у вас таково, что в понедельник, четверг и пятницу у вас 6 занятий, во вторник 8, в среду 7, а в выходные занятий нет.

Напишите это с помощью инструкции `если`. У вас получится такой пример.

```
метод Скрипт()
    пер ДеньНедели = 1
    пер Занятий = 6

    если ДеньНедели == 1 или ДеньНедели == 4 или ДеньНедели == 5
        Занятий = 6

    иначе если ДеньНедели == 2
        Занятий = 8

    иначе если ДеньНедели == 3
        Занятий = 7

    иначе если ДеньНедели == 6 или ДеньНедели == 7
        Занятий = 0
    ;
```

```
    Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
;
```

После инструкции `если` находится вывод в терминал. Получилось именно то, что нужно сделать для каждого дня. Вы узнаёте количество занятий и выводите.

Инструкция «для по»

Весь вопрос теперь в том, как выполнить эти строки скрипта для каждого дня недели. А для этого как раз и нужна инструкция `для по`. Она «снаружи» обрамляет тот фрагмент, который нужно выполнить несколько раз, и выглядит следующим образом. Сделайте такой же пример в своей конфигурации.

```
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        если ДеньНедели == 1 или ДеньНедели == 4 или ДеньНедели == 5
            Занятий = 6

        иначе если ДеньНедели == 2
            Занятий = 8

        иначе если ДеньНедели == 3
            Занятий = 7

        иначе если ДеньНедели == 6 или ДеньНедели == 7
            Занятий = 0
        ;

        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;
```

В результате в терминал будут выведены следующие строки.

```
В 1-й день 6 занятий
В 2-й день 8 занятий
В 3-й день 7 занятий
В 4-й день 6 занятий
В 5-й день 6 занятий
В 6-й день 0 занятий
В 7-й день 0 занятий
```

Обратите внимание, что инструкция объявления переменной **Занятий** вынесена за пределы цикла, за пределы инструкции `для по`. Внутри цикла только меняется значение этой переменной.

Инструкция `для по` работает следующим образом. Когда исполнение скрипта доходит до неё, переменной цикла присваивается начальное значение. В вашем случае переменная цикла — **ДеньНедели**, а начальное значение для неё **1**.

Затем начинают выполняться те инструкции, которые расположены внутри цикла. Когда исполнение доходит до строки точки с запятой «;», завершающей инструкции цикла, переменная цикла автоматически увеличивается на единицу и все возвращается в начало цикла.

Если переменная цикла не превысила своё конечное значение (в вашем случае 7), инструкции внутри цикла выполняются ещё раз. Если переменная цикла превысила своё конечное значение, исполнение переходит дальше, к тем строкам, которые расположены после точки с запятой «;», завершающей цикл.

Запустите пример в режиме отладки и, двигаясь по шагам, посмотрите, как это работает. День недели будет последовательно изменяться от 1 до 7. В аргументе метода `Консоль.Записать()` у вас будут получаться строки с количеством занятий в каждый из дней.

Это не единственная форма инструкции цикла. Есть ещё две другие формы этой инструкции. С ними вы познакомитесь позже: для из *(122)*, пока *(128)*.

27. Задание простое

Выведите в терминал количество дней в каждом месяце из второй декады в виде **4-й месяц 30 дней**. Используйте переменную **НомерМесяца**. Решение смотрите здесь *(297)*.

28. Задание сложное

Перечислите школьные оценки от самой лучшей к самой худшей. Результат выведите в терминал. Решение смотрите здесь *(297)*.

Инструкция «выбор»

На предыдущем занятии вы написали цикл, в котором в инструкции `если` перебирали разные дни недели и устанавливали количество занятий.

```
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        если ДеньНедели == 1 или ДеньНедели == 4 или ДеньНедели == 5
            Занятий = 6

        иначе если ДеньНедели == 2
            Занятий = 8

        иначе если ДеньНедели == 3
            Занятий = 7

        иначе если ДеньНедели == 6 или ДеньНедели == 7
            Занятий = 0
        ;

        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;
```

Инструкция `если` позволяет задать самые разные сложные условия для самых разных переменных и выражений. Однако есть большое количество случаев, когда нужно проанализировать всего одну переменную или выражение на довольно простые условия. Тогда удобнее использовать инструкцию `выбор`. В этом примере она хорошо подходит и помогает сделать его более «читабельным».

При использовании инструкции `выбор` этот пример будет выглядеть следующим образом.

```
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        выбор ДеньНедели
        когда 1, 4, 5
            Занятий = 6
        когда 2
            Занятий = 8
        когда 3
            Занятий = 7
        когда 6, 7
            Занятий = 0
        ;

        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;
```

В этом примере инструкция `выбор` работает следующим образом.

Ключевым словом `выбор` задаётся ***выражение-выбора***. В примере это переменная **ДеньНедели**.

Затем последовательно, для каждого условия `когда`, в котором указано ***выражение-сравнения***, выполняется логическая операция ***выражение-выбора*** `==` ***выражение-сравнения***.

Если результат операции — `Истина`, выполняются операторы в этой ветви и затем операторы, расположенные после инструкции `выбор`. Если результат операции — `Ложь`, анализируется следующее условие `когда`.

Если нужно сравнить ***выражение-выбора*** сразу с несколькими выражениями сравнения, эти выражения перечисляются через запятую. Тогда перечисленные условия будут объединены по `или`. Например, так происходит в первом `когда`.

```
выбор ДеньНедели
когда 1, 4, 5
    Занятий = 6
когда 2
```

Если нужно не просто сравнить на равенство, а выполнить другую логическую операцию, нужная операция указывается после `когда` перед выражением выбора. Например, в последнем `когда` будет выполнена логическая операция **Сравнение на строгое больше**.

```
выбор Значение
когда 1
    Результат = "1"
когда 2, 3
    Результат = "2 или 3"
когда > 4
    Результат = "больше 4"
иначе
    Результат = "все остальное: " + Значение
;
```

Подробнее про инструкцию выбор читайте в руководстве разработчика.

Методы

Общая информация

Продолжите исследовать и улучшать ваш пример с количеством занятий в каждый из дней. Посмотрите на ту его часть, которая, зная день недели, узнает вам количество занятий.

скрипт.sbsl

```
▶ Выполнить | Отладить
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        выбор ДеньНедели
        когда 1, 4, 5
            Занятий = 6
        когда 2
            Занятий = 8
        когда 3
            Занятий = 7
        когда 6, 7
            Занятий = 0
        ;

        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;
```

В вашем примере она выглядит довольно просто, но в реальном скрипте это может быть большой и сложный алгоритм. Он по сложным правилам, в зависимости от многих условий решает, сколько будет занятий.

Представьте себе, что этот алгоритм вам нужно использовать не только здесь, где он написан, но и в каком-то другом месте вашего скрипта. Только там, например, вам нужно просто вычислить, сколько занятий будет в какой-то один определённый день, который вы заранее не знаете.

Что делать? В том, другом, месте заново ещё раз писать весь этот алгоритм? Это нехорошо.

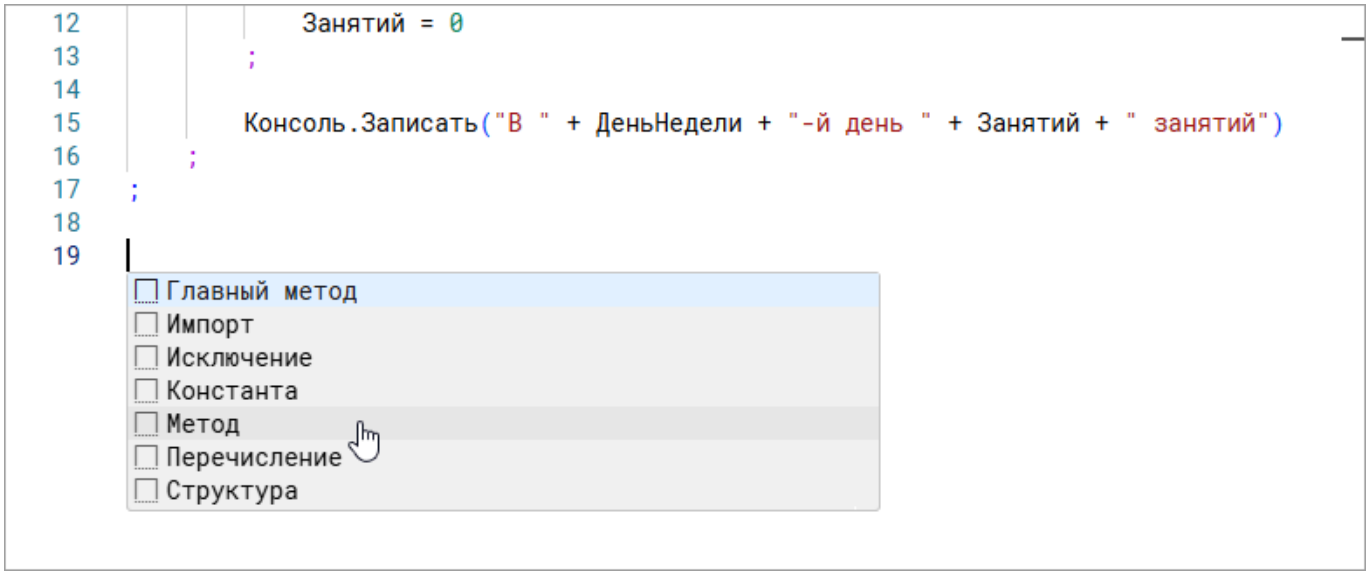
А если таких мест не одно, а несколько? А если вы потом захотите изменить этот алгоритм? Придётся изменять его во всех местах, где он написан? Тогда легко допустить ошибку и что-то пропустить.

Но выход есть! Нужно просто оформить этот алгоритм в виде метода. Тогда он будет написан у вас в единственном месте, а использовать его вы сможете в разных местах своего скрипта.

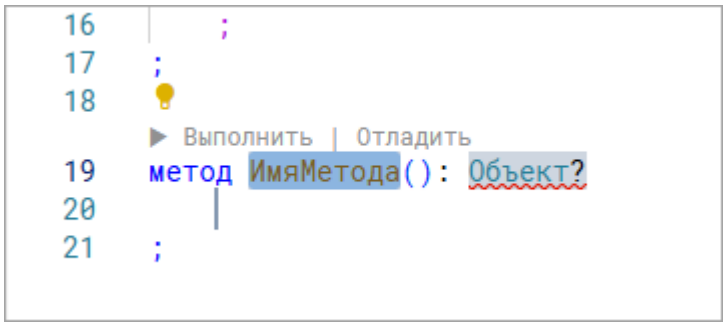
Определение метода

Метод — это не инструкция. Это просто специальная форма записи нескольких инструкций вместе. Это фрагмент программного кода, к которому можно обратиться из другого места скрипта.

Чтобы не было скучно, воспользуйтесь контекстной подсказкой. Установите курсор парой строк ниже последней точки с запятой «;» и нажмите **Ctrl+Пробел**. Нажмите **Метод**.



Среда разработки «1С:Предприятие.Элемент» вставит в ваш скрипт заготовку метода.



Теперь перетащите внутрь этого метода всю инструкцию `выбор`. Используйте форматирование, чтобы среда разработки сама расставила синтаксические отступы. У вас получится такой текст.

```
метод Скрипт()
    для ДеньНедели = 1 по 7

        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;

метод ИмяМетода(): Объект?
    выбор ДеньНедели
    когда 1, 4, 5
        Занятий = 6
    когда 2
        Занятий = 8
    когда 3
        Занятий = 7
    когда 6, 7
        Занятий = 0
    ;
;
```

То, что находится между строкой **метод ИмяМетода(): Объект?** и последней точкой с запятой «;», называется *телом метода*. Это те инструкции, которые будут выполнены при вызове метода.

Первая строка — **метод ИмяМетода(): Объект?** — это *объявление метода*. Им вы сейчас и займётесь, потому что пока у вас есть только заготовка.

Объявление метода всегда начинается с ключевого слова `метод`. За ним, через пробел, нужно указать имя этого метода. Имя вы придумываете сами. Оно нужно для того, чтобы вызвать этот метод из другого места. Назовите метод **СколькоЗанятий**.

```
метод СколькоЗанятий(): Объект?
    выбор ДеньНедели
...
```

После имени, без пробела, всегда следуют круглые скобки — «(» и «)». В них может быть что-то написано, а может и не быть. Это зависит от того, нужны вашему методу специальные, особенные данные для работы или нет.

В вашем случае метод должен сказать, сколько занятий в тот или иной день недели. Поэтому такие специальные данные ему нужны. Ему нужно сказать, какой именно день недели вас интересует.

Посмотрите на тело метода. Метод ожидает, что этот день недели будет находиться в переменной **ДеньНедели**. То есть значение, которое будет в этой переменной, метод должен получить откуда-то «снаружи».

Чтобы среда разработки это поняла, напишите имя переменной **ДеньНедели** внутри круглых скобок и укажите, что она должна иметь тип `Число`.

```
метод СколькоЗанятий(ДеньНедели: Число): Объект?
    выбор ДеньНедели
...
```

Как вы помните, «1С:Элемент» использует статическую типизацию и типы переменных, параметров и возвращаемых значений методов должны быть описаны в явном виде.

Теперь среда разработки знает, что значение для переменной **ДеньНедели** будет получено в тот момент, когда вы вызовете этот метод. Если это сейчас непонятно, не расстраивайтесь. Чуть позже вы увидите, как это происходит.

Такие переменные, написанные в объявлении метода в скобках после имени метода, называются *параметрами метода*. Через параметры тело метода получит доступ к значениям, которые вы предоставите методу во время его вызова. Статическая типизация позволяет отслеживать возможные ошибки ещё в момент написания скрипта. Если вдруг вы ошибётесь и решите передать в этот метод значение, например, типа `Строка`, среда разработки сразу увидит это и сообщит вам об ошибке.

У вашего метода всего один параметр — **ДеньНедели**. Другую переменную, **Занятий**, он создаст сам в процессе работы. Поэтому описание метода почти готово.

Единственное, что осталось сделать, это указать, значение какого типа будет возвращать ваш метод. Возвращать он будет переменную **Занятий**, а она имеет тип `Число`. Поэтому в конце объявления метода, после закрывающей круглой скобки «)», вместо **: Объект?** нужно написать описание типа `Число`.

```
метод СколькоЗанятий(ДеньНедели: Число): Число
    выбор ДеньНедели
...
```

Теперь переместитесь в конец метода. После того, как выполнится инструкция **выбор**, и до того, как закончится исполнение метода **СколькоЗанятий**, он должен вам вернуть значение переменной **Занятий**. Поэтому в начале метода нужно объявить эту переменную, **пер Занятий: Число**, а здесь, в конце, использовать ключевое слово `возврат` — написать **возврат Занятий**.

```
метод СколькоЗанятий(ДеньНедели: Число): Число
    пер Занятий: Число
    выбор ДеньНедели
    когда 1, 4, 5
        Занятий = 6
    когда 2
        Занятий = 8
    когда 3
        Занятий = 7
    когда 6, 7
        Занятий = 0
    ;
    возврат Занятий
;
```

Вызов метода

Вызов метода — это место в тексте скрипта, где исполнение кода передаётся в тело метода: указывается имя метода, указываются передаваемые ему значения (если такие есть), используется результат работы метода (если метод что-то возвращает).

Теперь вам нужно вызвать этот метод в том месте, в котором нужно. А в котором нужно? Наверное, вы уже догадались. В том самом месте, откуда вы «утащили» инструкцию `выбор`.

Чтобы вызвать метод **СколькоЗанятий**, нужно написать его имя, а в скобках — выражение или переменную. Значение этого выражения или переменной будет передано в метод. Такое значение, которое указывается во время вызова метода, называется *аргументом*.

Поскольку метод вернёт вам количество занятий, вызов метода нужно написать в инструкции присваивания. Надо присвоить это значение той переменной, которая вам нужна, то есть переменной **Занятий**: **Занятий = СколькоЗанятий(ДеньНедели)**.

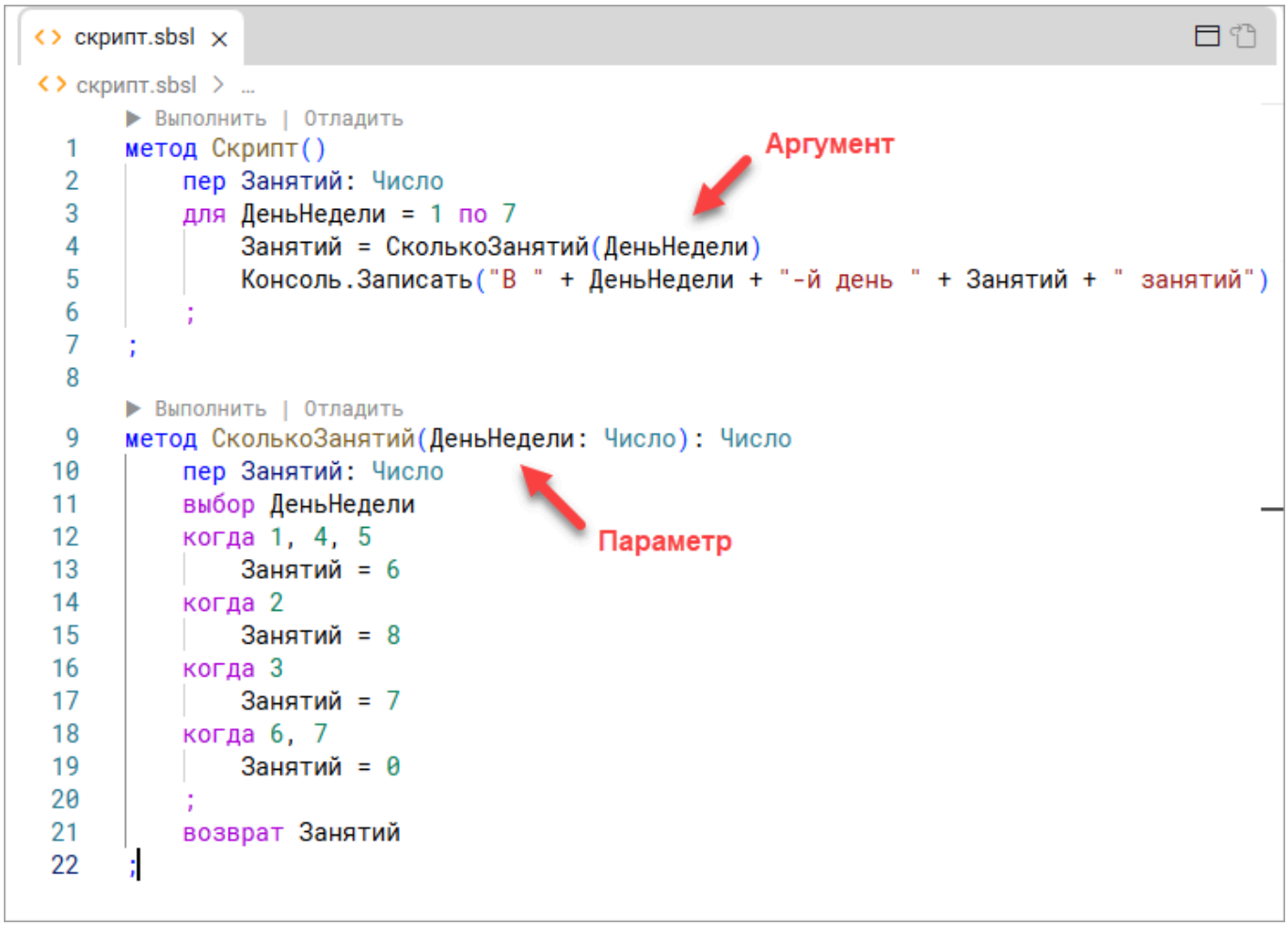
```
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        Занятий = СколькоЗанятий(ДеньНедели)
        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;
```

Как будет работать эта инструкция? Сначала, как вы знаете, «1С:Элемент» вычислит выражение, которое находится справа от знака равенства. То есть он вызовет метод **СколькоЗанятий()**. В этот метод он передаст значение переменной **ДеньНедели**.

В этом месте — там, где метод вызывается, — значения, указанные в скобках, называются аргументами. То есть это то, что на самом деле будет передано в метод.

Результатом вычисления выражения справа от знака равенства будет значение, которое вернёт ваш метод. Это значение будет присвоено переменной **Занятий**. После чего вы используете эту переменную, чтобы сформировать сообщение в терминал.

Чтобы убедиться, что вы не допустили ошибки, посмотрите на листинг ниже. Ваш скрипт должен выглядеть так. Заодно ещё раз обратите внимание на то, что называется аргументом, а что — параметром.



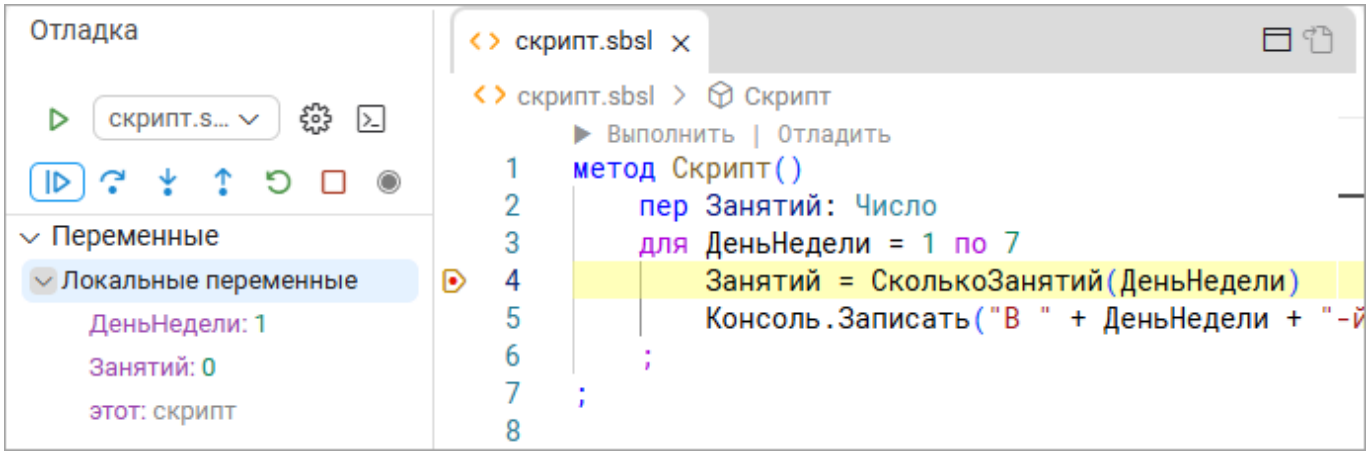

```
метод Скрипт()
    пер Занятий: Число
    для ДеньНедели = 1 по 7
        Занятий = СколькоЗанятий(ДеньНедели)
        Консоль.Записать("В " + ДеньНедели + "-й день " + Занятий + " занятий")
    ;
;

метод СколькоЗанятий(ДеньНедели: Число): Число
    пер Занятий: Число
    выбор ДеньНедели
    когда 1, 4, 5
```

```
        Занятий = 6
    когда 2
        Занятий = 8
    когда 3
        Занятий = 7
    когда 6, 7
        Занятий = 0
    ;
    возврат Занятий
;
```

Передача исполнения в метод

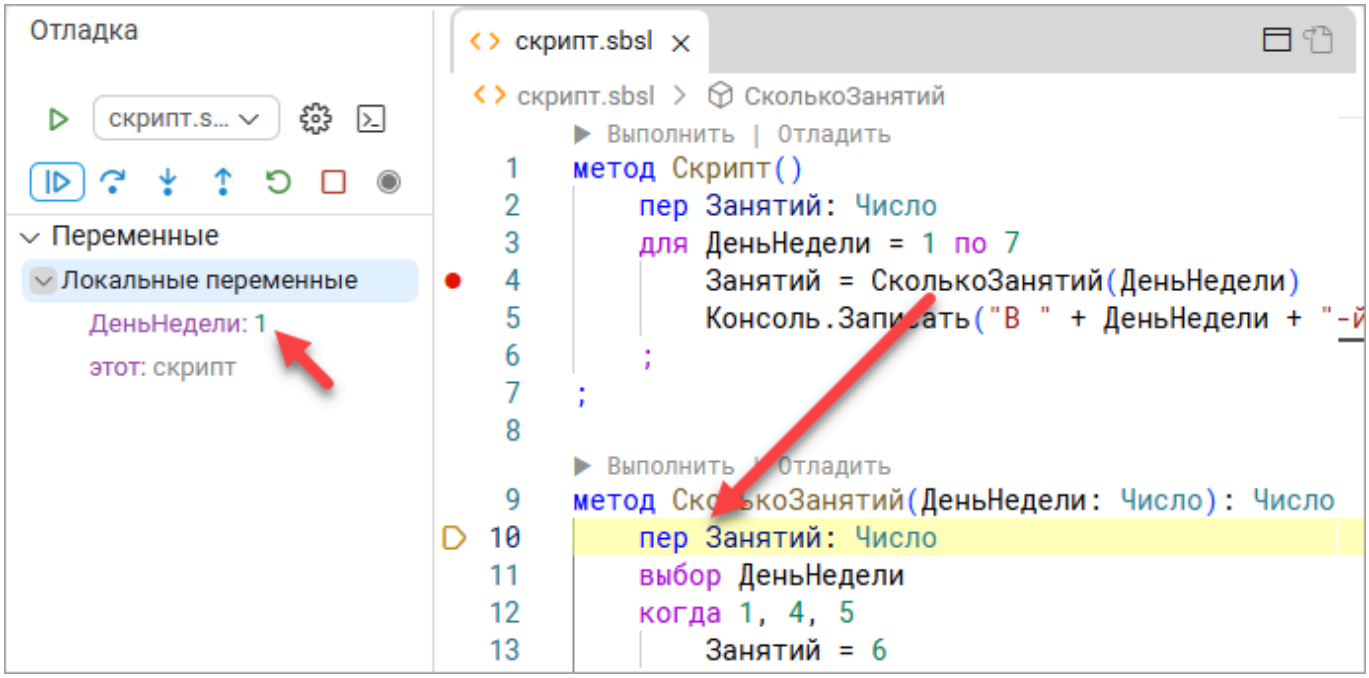
Теперь самое интересное! Установите точку останова в начале цикла и запустите скрипт в режиме отладки. Посмотрите, как это работает.

Сделайте один шаг. Движок «1С:Предприятие.Элемент Скрипт» остановится на инструкции присваивания. Он приготовится её исполнять.



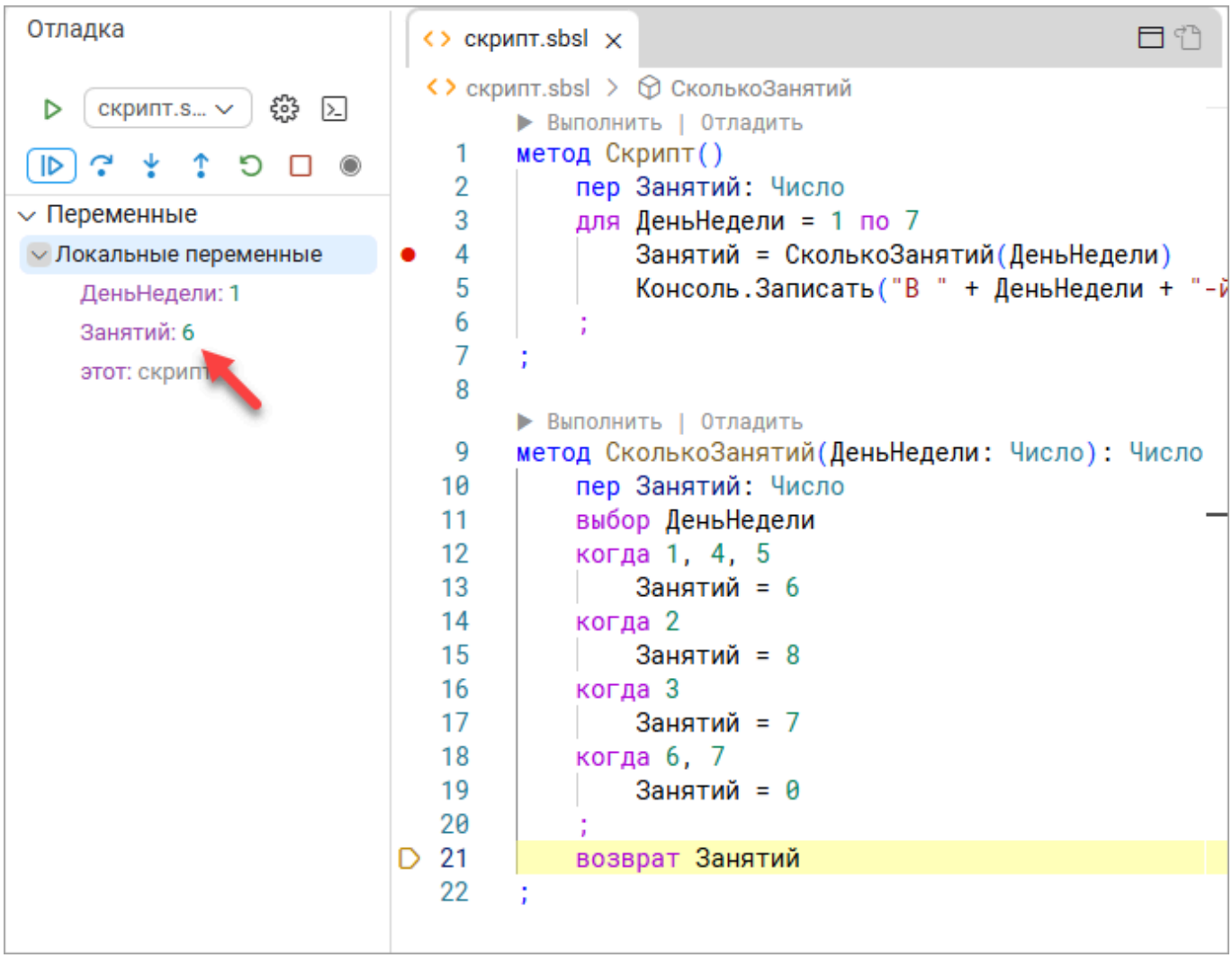
Это первый проход цикла, поэтому день недели равен 1.

Шагните ещё раз. Движок «провалится» в метод.



Обратите внимание, что движок передал сюда значение дня недели 1. Теперь есть все необходимые данные для того, чтобы выполнить инструкцию `если`.

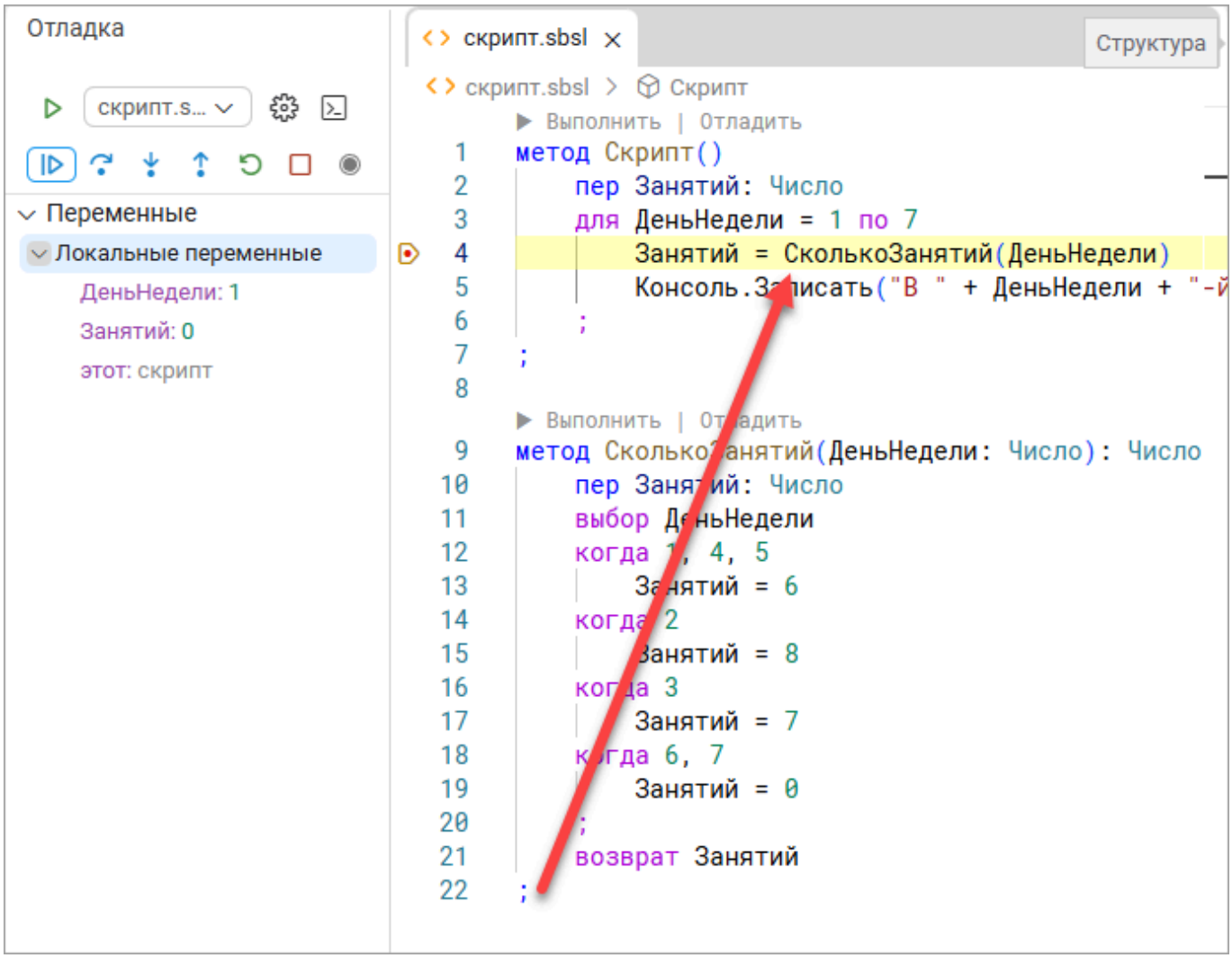
Сделайте ещё несколько шагов, чтобы движок остановился перед выполнением инструкции `возврат`.



Вы видите, какое значение метод собирается возвращать. Это значение **6**.

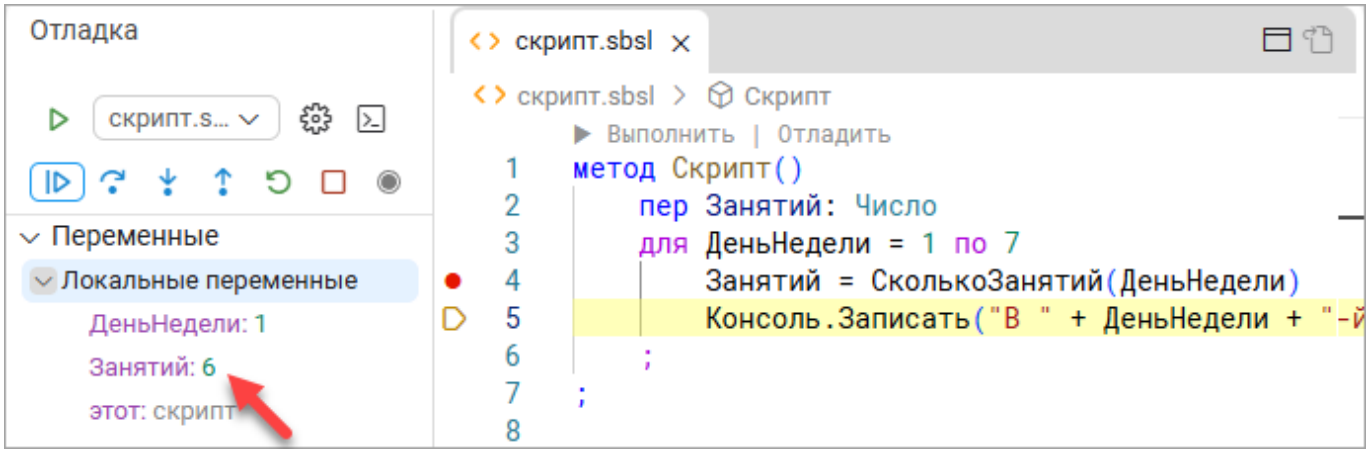
Сделав ещё шаг, вы перейдёте на последнюю точку с запятой «;». Это значит, что исполнение метода закончилось.

Шагнув ещё раз, вы вернётесь обратно к инструкции присваивания.

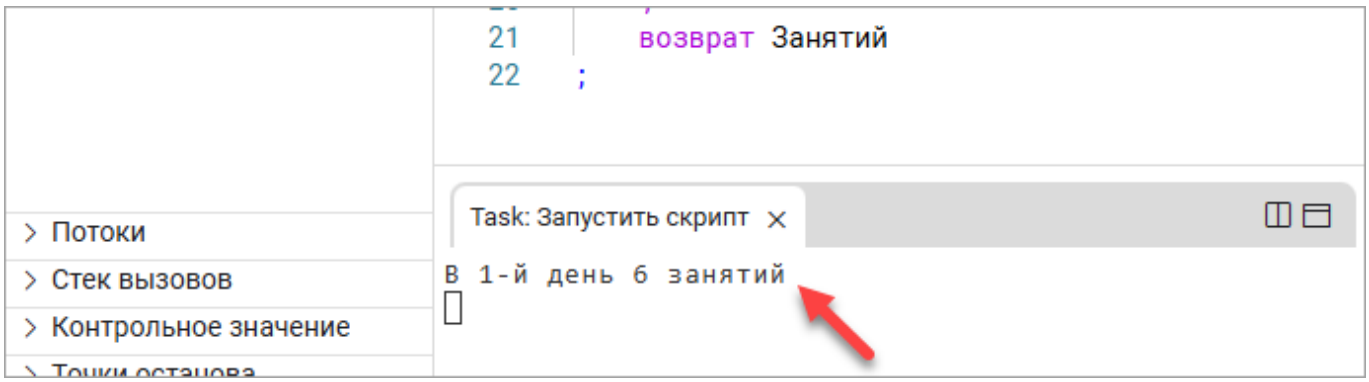


Сейчас у вас картинка очень похожа на то, что было в начале. Движок приготовился выполнить инструкцию присваивания. Однако если раньше движок собирался вычислить выражение, находящееся справа, то теперь он его уже вычислил. Сейчас он готов поместить получившееся значение в переменную.

Сделайте один шаг. Исполнение перейдёт к следующей строке. В переменной **Занятий** появится значение **6**. Это то значение, которое вернул метод.

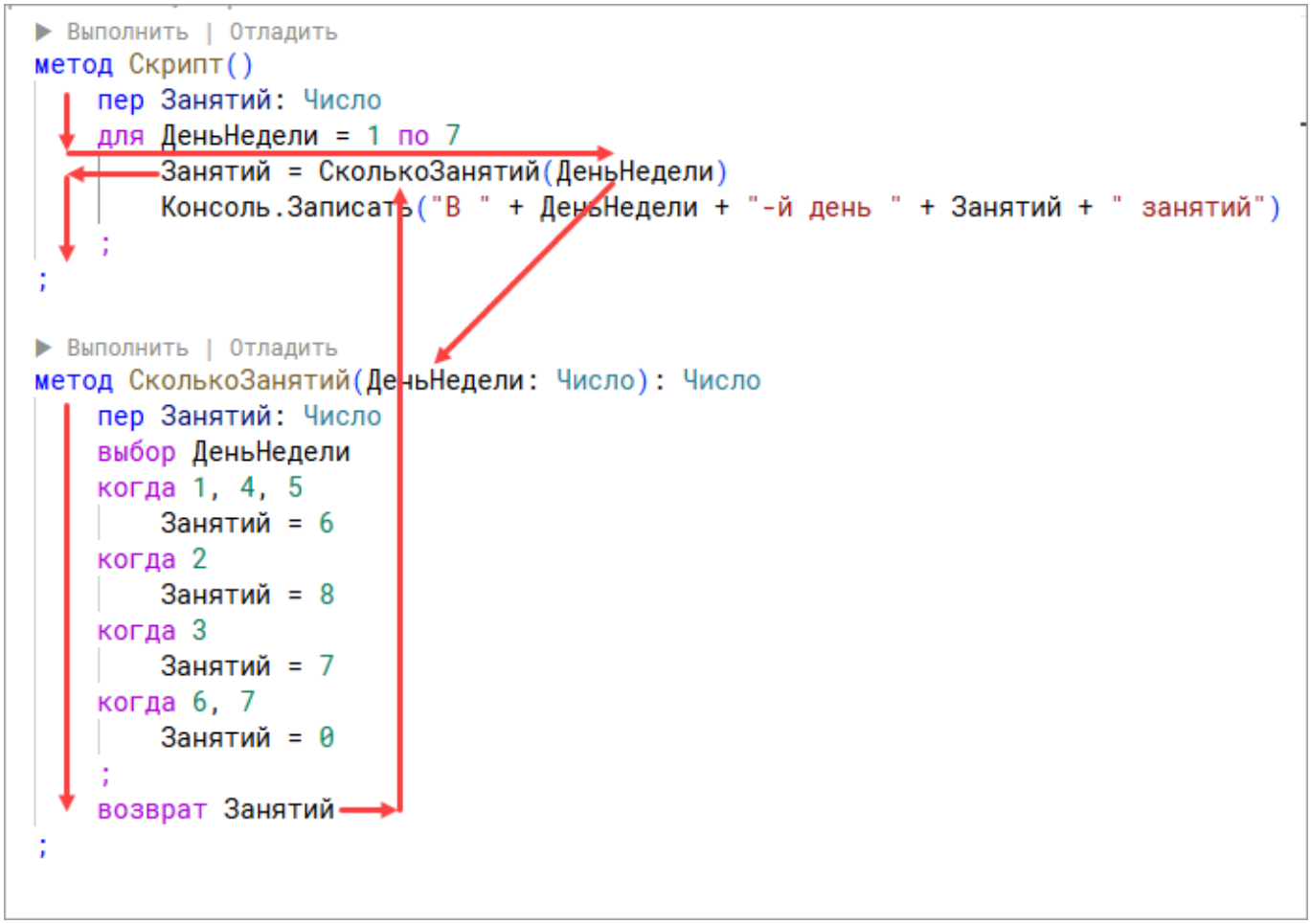


Если вы шагнёте ещё раз, то в терминал запишется строка.



После этого исполнение цикла продолжится, но день недели будет равен **2**. И так далее. Можете пройти несколько итераций цикла и посмотреть ещё раз, как работает этот пример.

Чтобы закрепить свои знания, посмотрите на рисунок. На нём стрелками показана последовательность исполнения «1С:Элемента» при вызове метода.



В тот момент, когда «1С:Элемент» встречает вызов метода, исполнение «проваливается» в метод. После того как метод выполнит инструкции, написанные в его теле, исполнение возвращается в ту точку, из которой был вызван метод. Только после этого исполнение «1С:Элемента» переходит к следующей инструкции.

Методы без возвращаемого значения

Методы могут не возвращать никакого значения и даже могут не получить никаких аргументов. Метод может просто выполнять те инструкции, которые написаны в его теле. Посмотрите на следующий пример.

```
метод Скрипт()
    ПредупредитьОПонедельнике()
;

метод ПредупредитьОПонедельнике()
    если ДатаВремя.Сейчас().ДеньНедели() == ДеньНедели.Воскресенье
        Консоль.Записать("Завтра в школу!")
    ;
;
```

Если метод ничего не возвращает, то в его объявлении после круглых скобок «()» не нужно никакое описание типа. Также внутри круглых скобок не нужны описания его параметров, если параметров нет.

Вызов метода без возвращаемого значения выглядит просто как имя метода и круглые скобки «()».

Примечание:

Подробнее о методах читайте в руководстве разработчика.

Примечание:

Работа с методами на углублённом уровне описана здесь *(207)*.

29. Задание простое

Для тестирования скрипта вам требуется длинная строка. Вводить её вручную неудобно. Напишите метод, в который вы будете передавать короткую строку-образец и количество раз, сколько эту строку нужно повторить. Возвращать метод будет одну строку, в которой строка-образец повторена указанное количество раз. Решение смотрите здесь *(298)*.

30. Задание простое

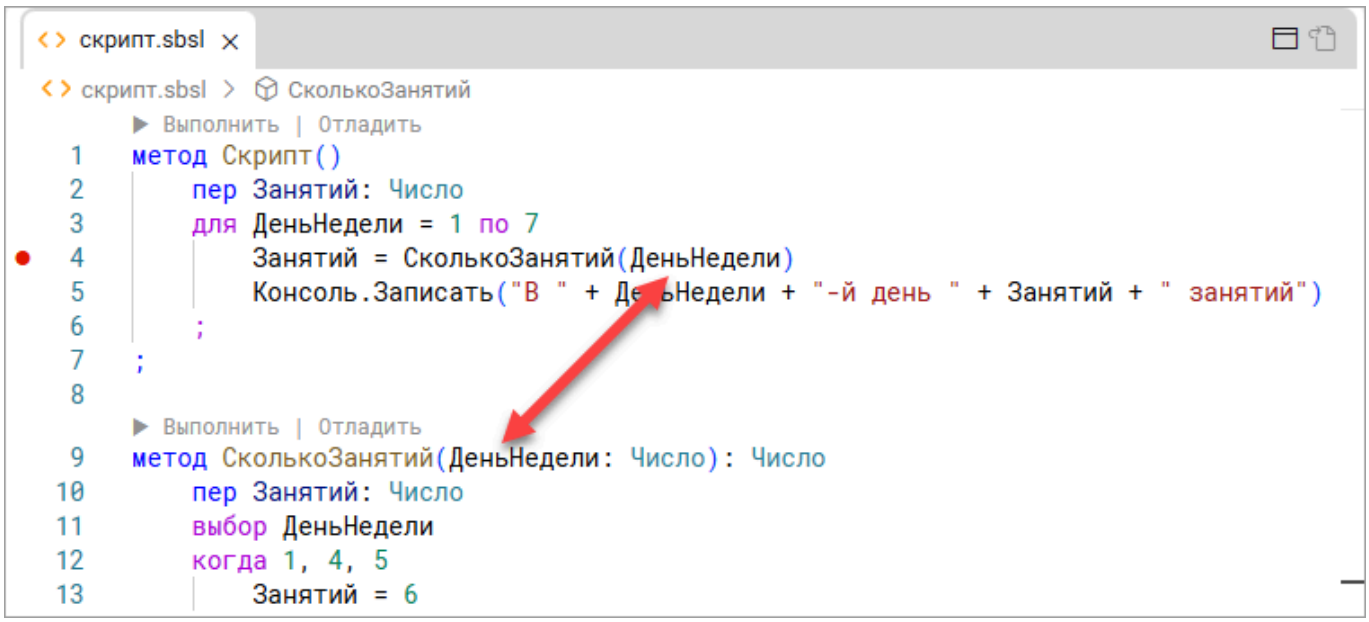
Напишите метод, который будет получать текущую «ДатуВремя» и выводить в терминал представление этой «ДатыВремени» в виде:

```
Сегодня 02.09.2025, Вторник
```

Чтобы выделить из ДатыВремени только дату, используйте свойство `ДатаВремя.Дата`. Решение смотрите здесь *(298)*.

Область видимости имён

Вы, наверное, обратили внимание на то, что в вашем примере имена аргумента и параметра совпали — **ДеньНедели**.



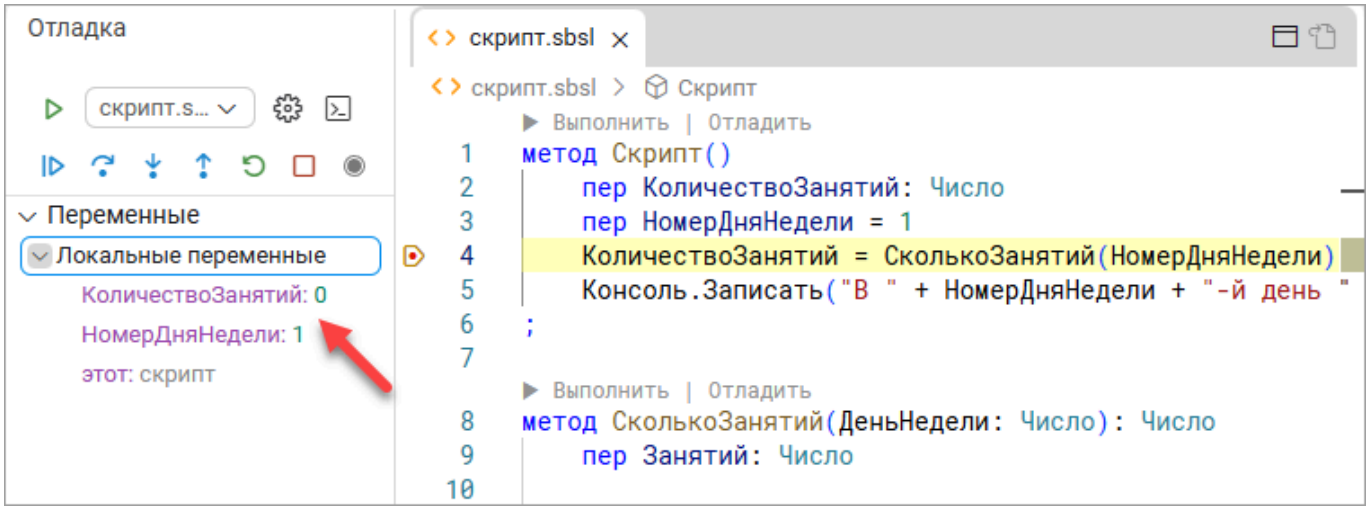
У вас это получилось случайно. Просто потому, что вы перетаскивали фрагмент скрипта из одного места в другое.

На самом деле имена аргументов и параметров обычно не совпадают. В этом нет необходимости.

Чтобы разобраться с этим, возьмите упрощённый пример. Здесь в методе **Скрипт()** убран цикл, а переменные переименованы в **КоличествоЗанятий** и **НомерДняНедели**. В методе **СколькоЗанятий()** оставлены только две ветки в инструкции `выбор`.

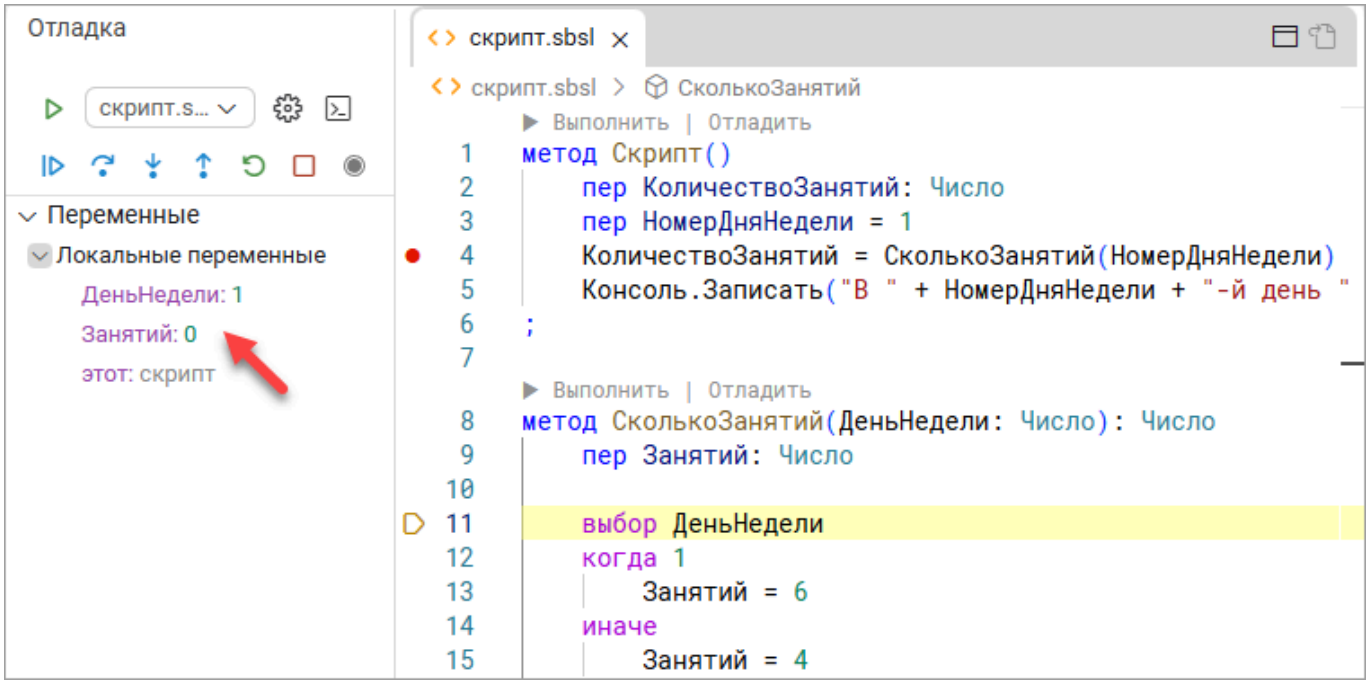
```
метод Скрипт()
    пер КоличествоЗанятий: Число
    пер НомерДняНедели = 1
    КоличествоЗанятий = СколькоЗанятий(НомерДняНедели)
    Консоль.Записать("В " + НомерДняНедели + "-й день " + КоличествоЗанятий + " занятий")
;

метод СколькоЗанятий(ДеньНедели: Число): Число
    пер Занятий: Число

    выбор ДеньНедели
    когда 1
        Занятий = 6
    иначе
        Занятий = 4
    ;

    возврат Занятий
;
```

Установите точку останова на инструкции, в которой вызывается метод **СколькоЗанятий()**. Запустите скрипт в режиме отладки и откройте локальные переменные. Они вам сейчас очень помогут разобраться с тем, что будет происходить.



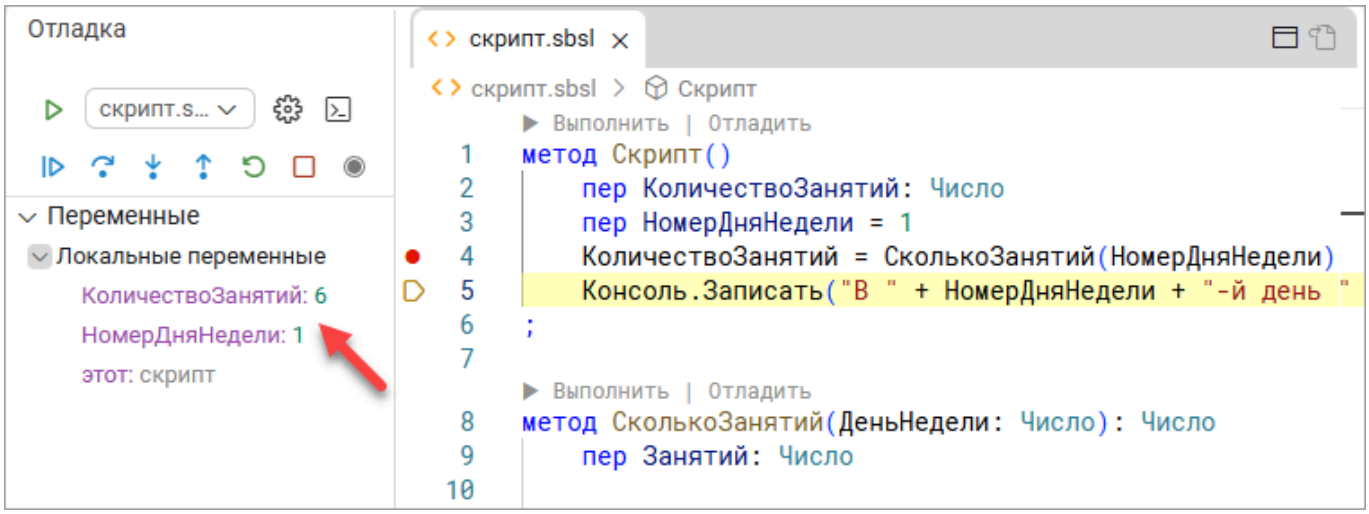
Вы находитесь в методе **Скрипт()**. Здесь вам доступны переменные **КоличествоЗанятий** и **НомерДняНедели**. Это те переменные, которые создаются и определяются в этом методе.

Теперь сделайте пару шагов, чтобы перейти в метод **СколькоЗанятий()**.



Обратите внимание на то, что вы видите в разделе локальных переменных: **КоличествоЗанятий** и **НомерДняНедели** исчезли. Зато вместо них появились **ДеньНедели** и **Занятий**, которые определены внутри этого метода.

Теперь, если вы пройдёте весь метод и вернётесь обратно, состав локальных переменных снова изменится.



Он станет таким, каким он был в самом начале. На что это похоже?

Это похоже на то, что вы ходите из одной комнаты в другую. Каждый метод — как отдельная комната. Например, гостиная и кухня.

Когда вы находитесь в гостиной, вы видите только то, что находится в гостиной. Диван, кресло, журнальный столик. Вы не видите то, что находится на кухне.

Если вы выйдете из гостиной и зайдёте на кухню, вы увидите то, что находится на кухне: обеденный стол, стулья, холодильник, духовку. И в это же время вы перестанете видеть, что находится в гостиной.

Для обозначения этой ситуации в «1С:Элементе» есть понятие *область видимости имён*. Область видимости — это часть текста скрипта, в пределах которой имя переменной связано с этой переменной. В одной области видимости не может существовать двух одинаковых имён.

Существует несколько типов областей видимости.

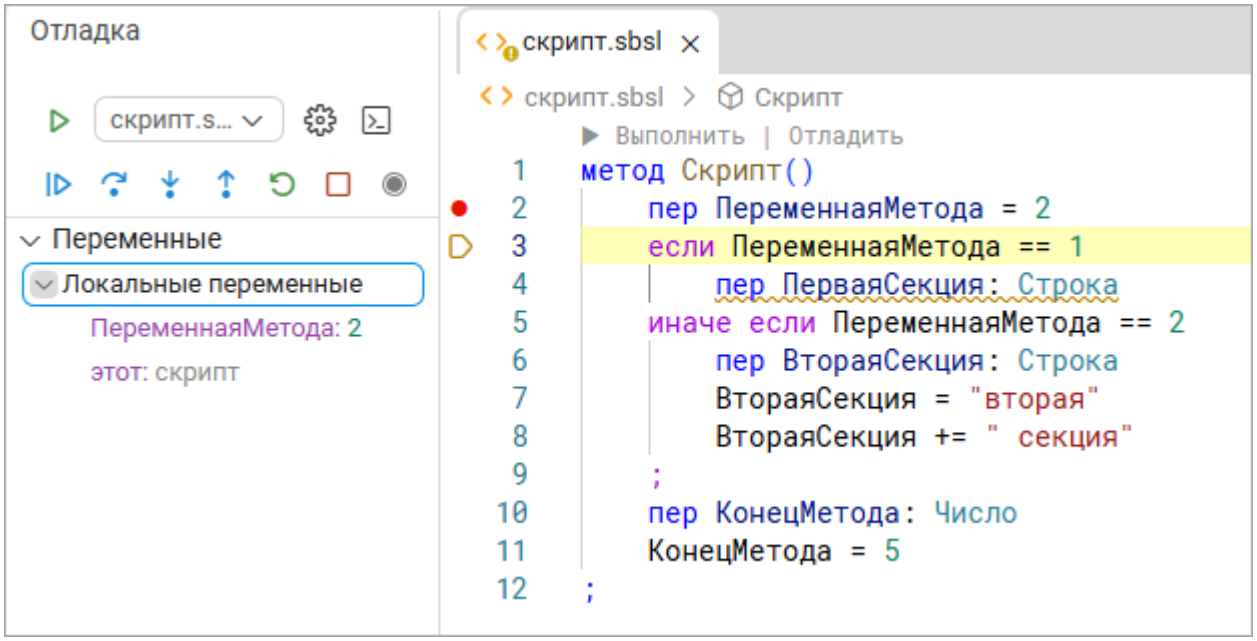
Область видимости скрипта. К этой области относятся, например, специальные переменные, константы, с которыми вы познакомитесь позже *(200)*. Они объявляются вне методов, в начале скрипта. Они доступны во всех методах, которые содержатся в скрипте.

Область видимости блока кода. К этому типу относятся два случая:

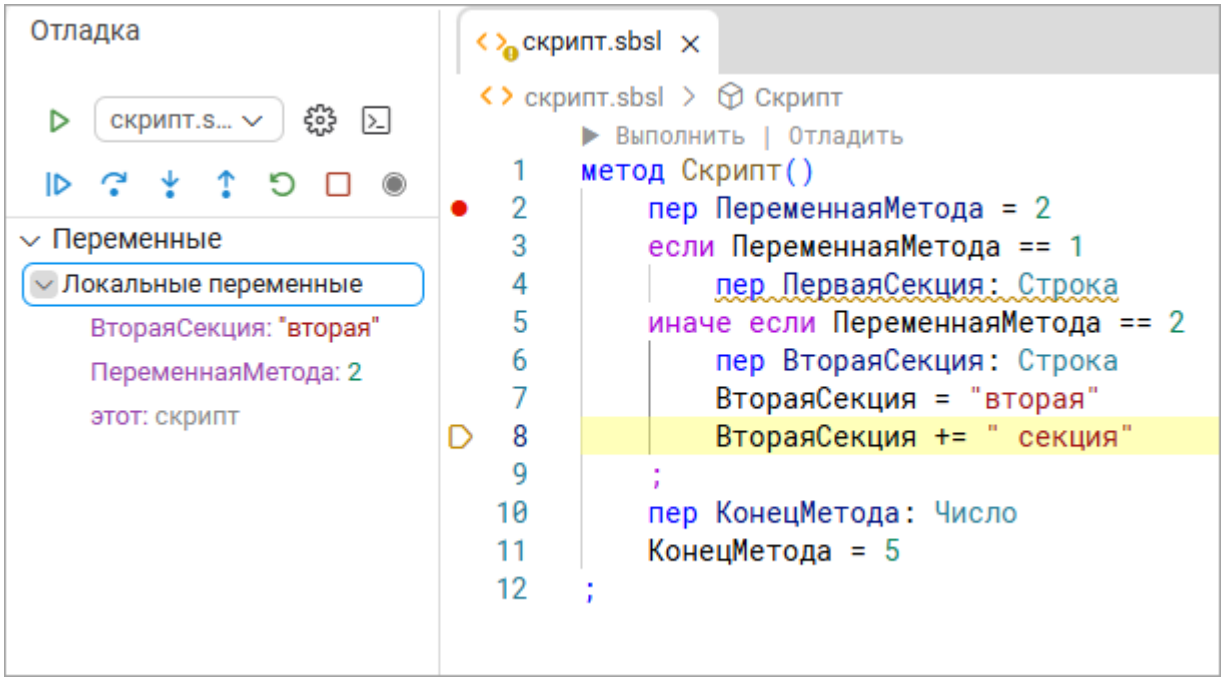
- Область видимости метода. Это как раз тот случай, который вы рассматриваете. Метод является блоком кода, и переменные, объявленные внутри метода, видимы от места объявления до конца метода.
- Область видимости внутри метода. Дело в том, что блоки кода образуются не только методами, но и более мелкими конструкциями: секциями инструкций `если`, телами циклов и ключевым словом `область`.

В рамках этой книги вы не будете рассматривать области видимости внутри методов. Однако для общего развития можете скопировать в свой скрипт следующий пример, пройти его по шагам и посмотреть, какие переменные в какие моменты времени вам видны.

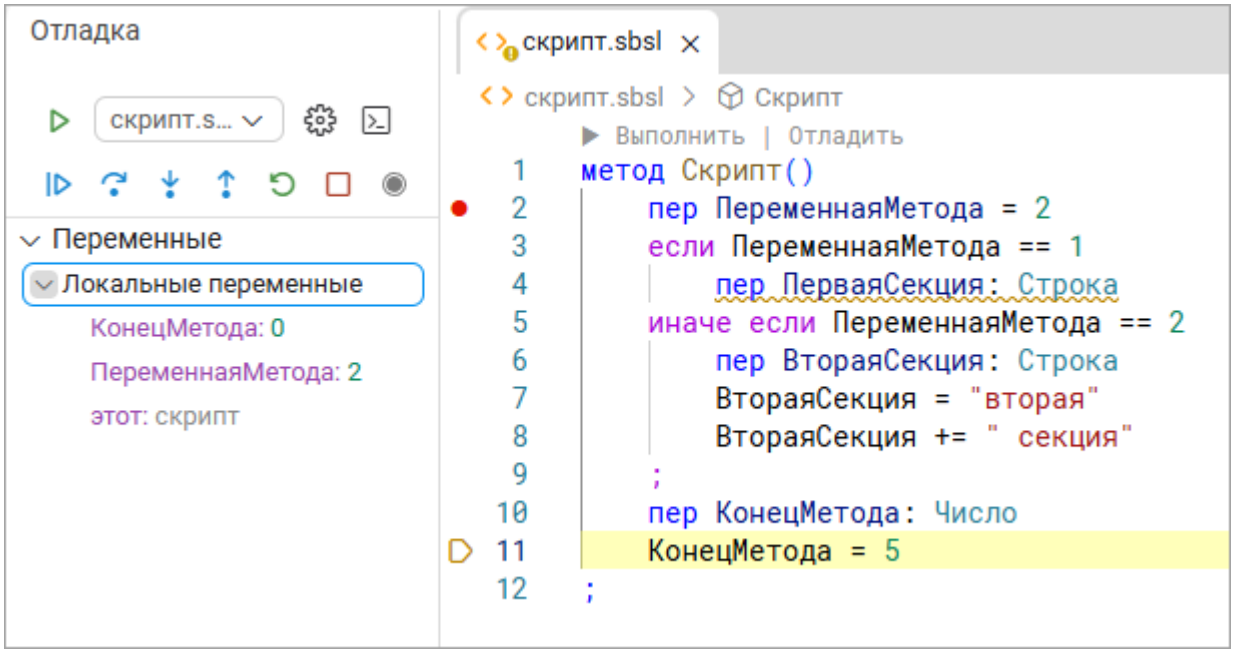
```
метод Скрипт()
    пер ПеременнаяМетода = 2
    если ПеременнаяМетода == 1
        пер ПерваяСекция: Строка
    иначе если ПеременнаяМетода == 2
        пер ВтораяСекция: Строка
        ВтораяСекция = "вторая"
        ВтораяСекция += " секция"
    ;
    пер КонецМетода: Число
    КонецМетода = 5
;
```



Перед выполнением инструкции `если` вы видите переменную **ПеременнаяМетода**. Её область видимости — до конца метода.



После того как вы зашли в ветку условия, вы видите переменную **ВтораяСекция**. Её область видимости — до конца этой ветки. Как только вы сделаете ещё один шаг, она исчезнет.



Под конец вы видите, что появилась ещё одна переменная, определённая в области видимости метода, — **КонецМетода**. Она будет видна также до конца метода.

> **Примечание:**
>
> Подробнее про области видимости имён читайте в руководстве разработчика.

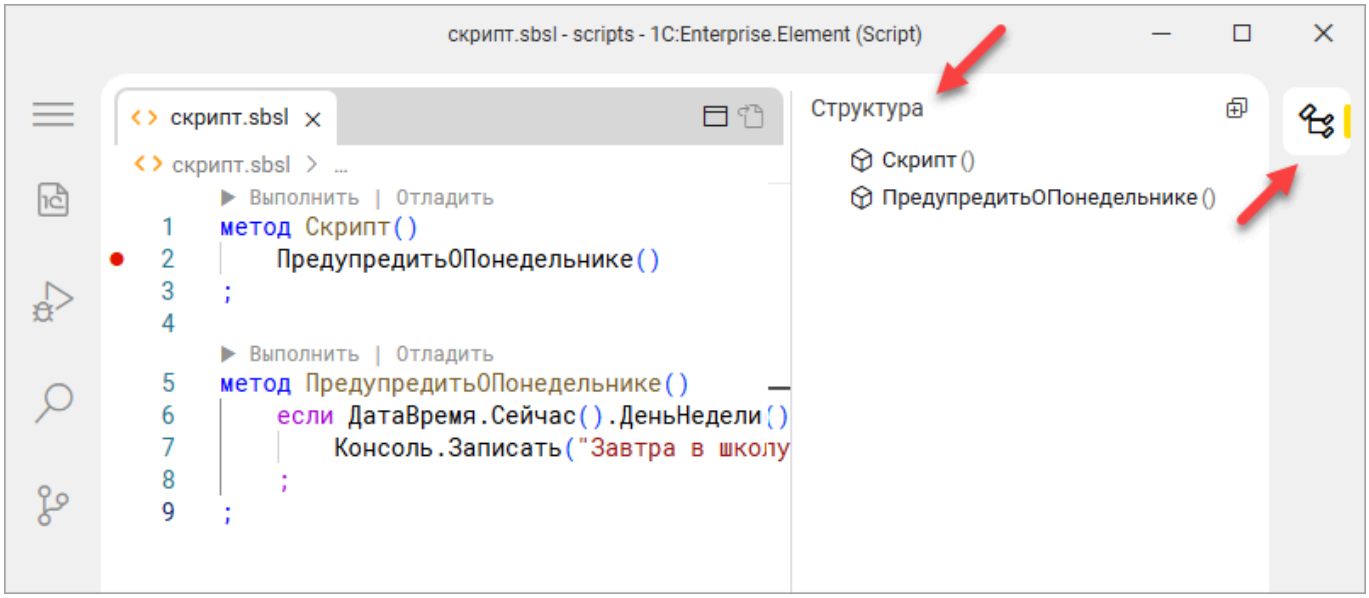
Чтение и отладка методов

Панель «Структура»

Скрипты могут быть большими, а методы могут быть расположены в разных частях скрипта. Совсем не обязательно объявление метода будет находиться где-то рядом с тем местом, откуда он вызывается. Это так только в вашем примере.

Чаще всего вызов метода и его объявление находятся далеко друг от друга. Как в этом случае быстро найти объявление метода, чтобы посмотреть, какие действия он выполняет?

Для этого есть панель **Структура**. Она расположена с правой стороны экрана.



Если эта панель свёрнута, то нажмите на значок на правой панели действий, чтобы развернуть её.

Если панель **Структура** совсем закрыта, вы можете открыть её. Для этого нажмите **Главное меню > Вид > Структура**.

В панели **Структура** перечислены все методы, существующие в вашем скрипте. Если вы нажмёте на один из них, то в редакторе среда разработки «1С:Предприятие.Элемент» выделит имя этого метода в его объявлении.

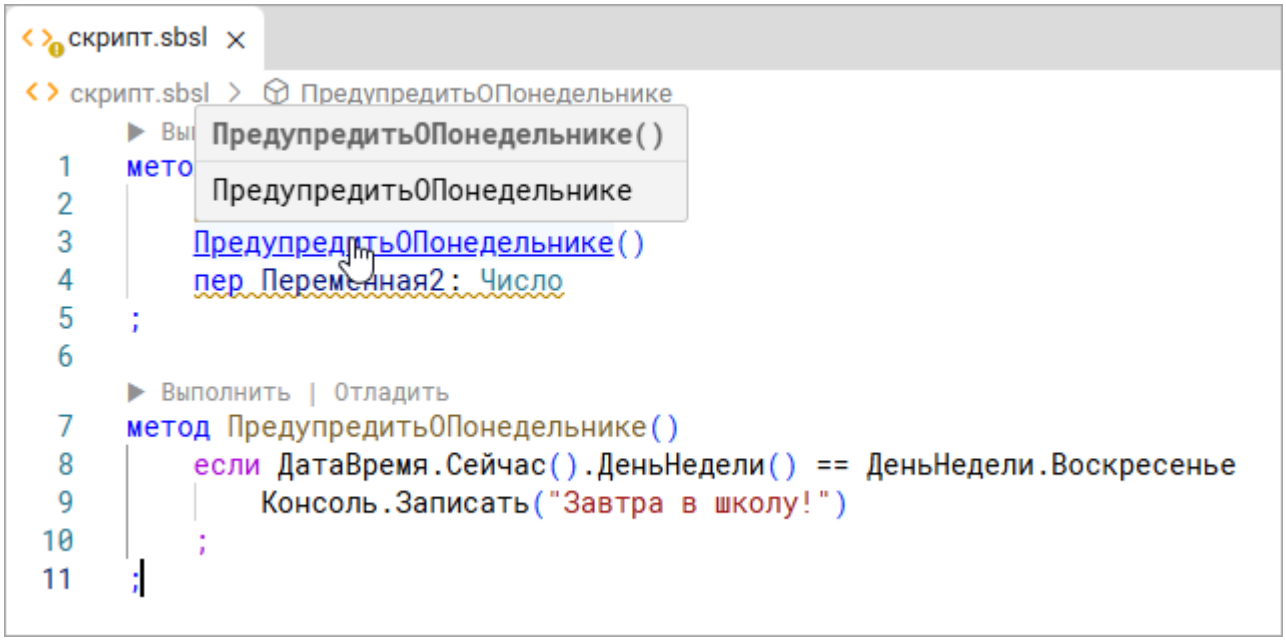
Переход к объявлению метода

Может так случиться, что методов в вашем скрипте очень много. Искать глазами в панели **Структура** нужный метод становится трудно.

В этом случае на помощь вам придёт команда перехода к объявлению метода. Скопируйте в свой скрипт следующий текст.

```
метод Скрипт()
    пер Переменная1: Строка
    ПредупредитьОПонедельнике()
    пер Переменная2: Число
;

метод ПредупредитьОПонедельнике()
    если ДатаВремя.Сейчас().ДеньНедели() == ДеньНедели.Воскресенье
        Консоль.Записать("Завтра в школу!")
    ;
;
```

Нажмите **Ctrl** и, не отпуская её, подведите курсор к вызову метода. Вызов превратится в гиперссылку. Нажмите на неё.



«1С:Элемент» установит курсор на объявление метода **ПредупредитьОПонедельнике()**.

Чтобы вернуться обратно, нажмите **Alt+Влево** два раза. Первый раз курсор встанет на конец метода, а во второй раз вернётся на вызов этого метода.

> **Возможная проблема:**
>
> Нажимайте клавишу **Alt**, расположенную слева от пробела.

Есть и другой способ. Установите курсор на вызов метода и в контекстном меню нажмите **Перейти к объявлению**.

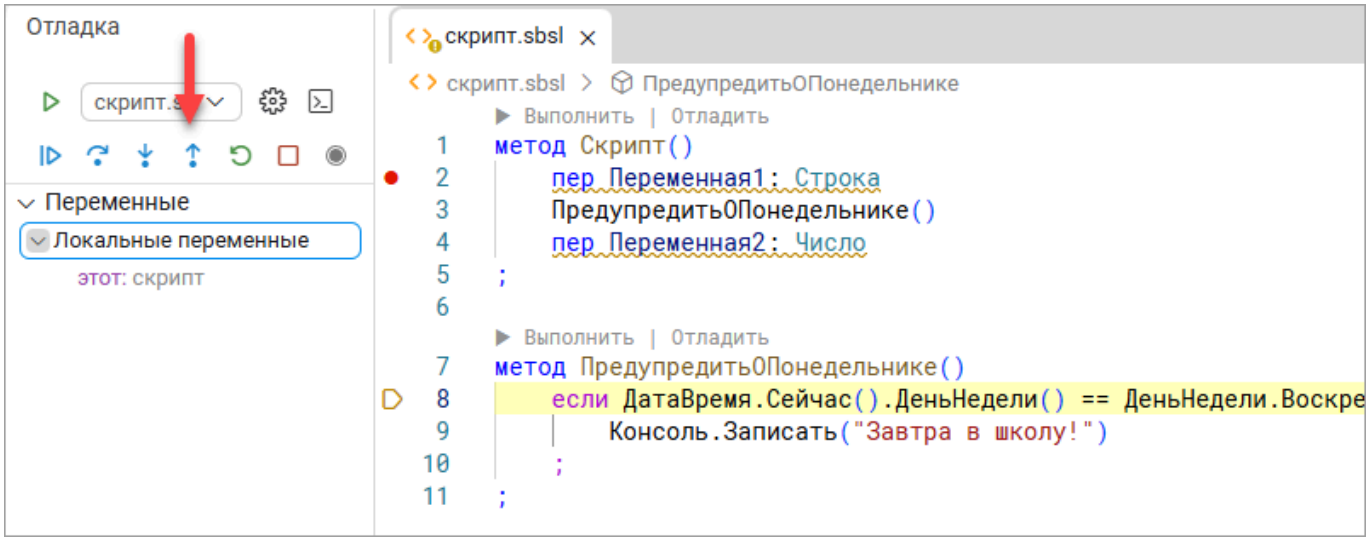
Шаг с выходом

Пара интересных приёмов связана с отладкой методов. Вы умеете выполнять отладку по шагам и используете для этого клавишу **F11**. Она позволяет вам останавливаться на каждой инструкции, которая исполняется.

Однако это не всегда удобно. Бывают случаи, когда хочется быстрее выйти из метода, не проходя по шагам каждую оставшуюся инструкцию.

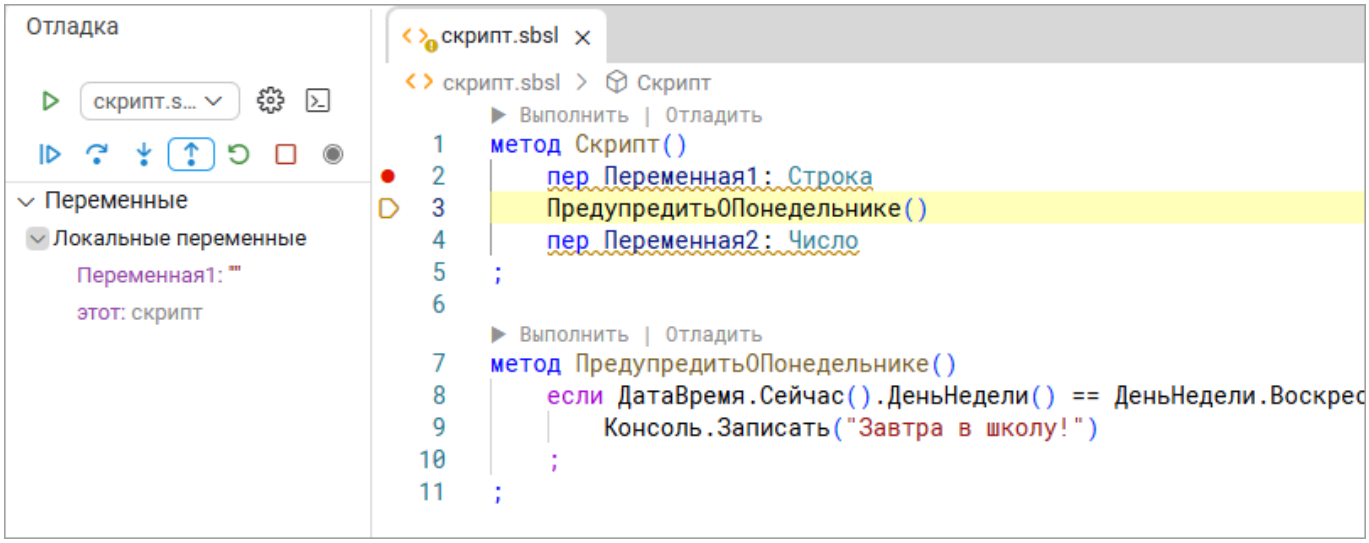
Представьте, что вы отлаживаете скрипт, чтобы найти ошибку. Вы зашли в метод, посмотрели на его текст и поняли, что ошибка не в нём, а где-то дальше. Поэтому нет смысла проходить весь метод по шагам. Нужно вернуться к тому месту, откуда метод вызывался.

Смоделируйте эту ситуацию. В примере установите точку останова на второй строке, запустите отладку и по шагам дойдите до инструкции `если`, расположенной в методе.



Теперь выполните команду **Шаг с выходом** — она расположена в представлении **Отладка** в командной панели, четвёртая по счёту. Также вы можете вызвать эту команду сочетанием клавиш **Shift+F11** или нажав **Главное меню > Запустить > Шаг с выходом**.

В результате выполнения этой команды инструкции, содержащиеся в методе далее, будут исполнены без остановки. Остановка произойдёт тогда, когда исполнение вернётся к той строке, в которой метод был вызван.

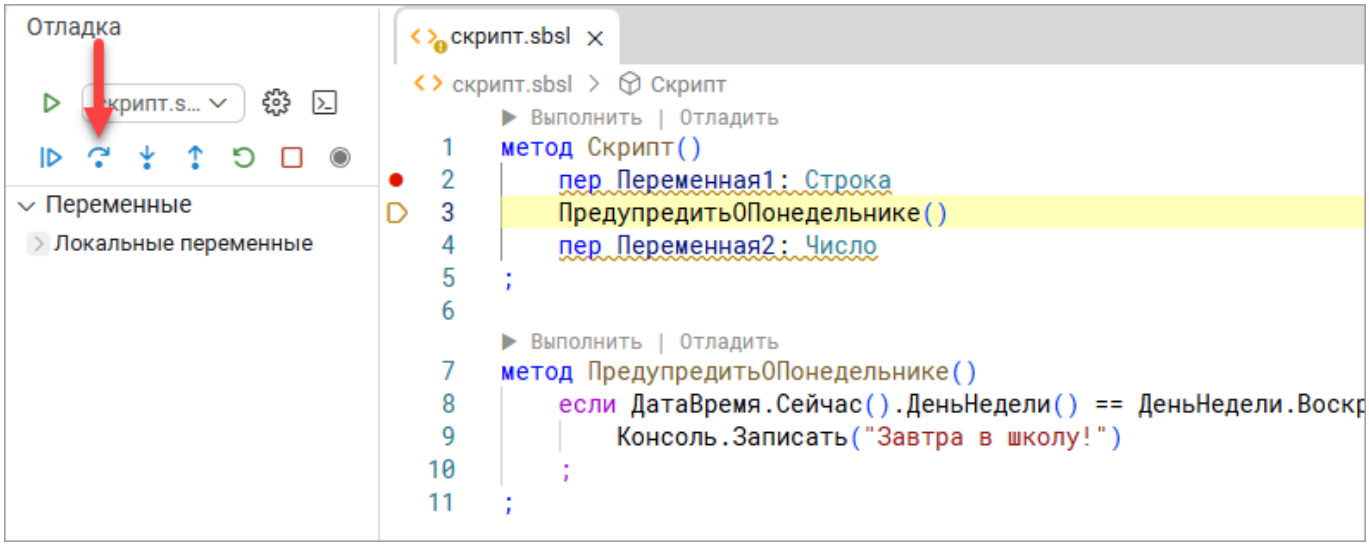


Шаг с обходом

Теперь рассмотрите другую ситуацию. Перезапустите отладку и остановитесь на вызове метода **ПредупредитьОПонедельнике()**.

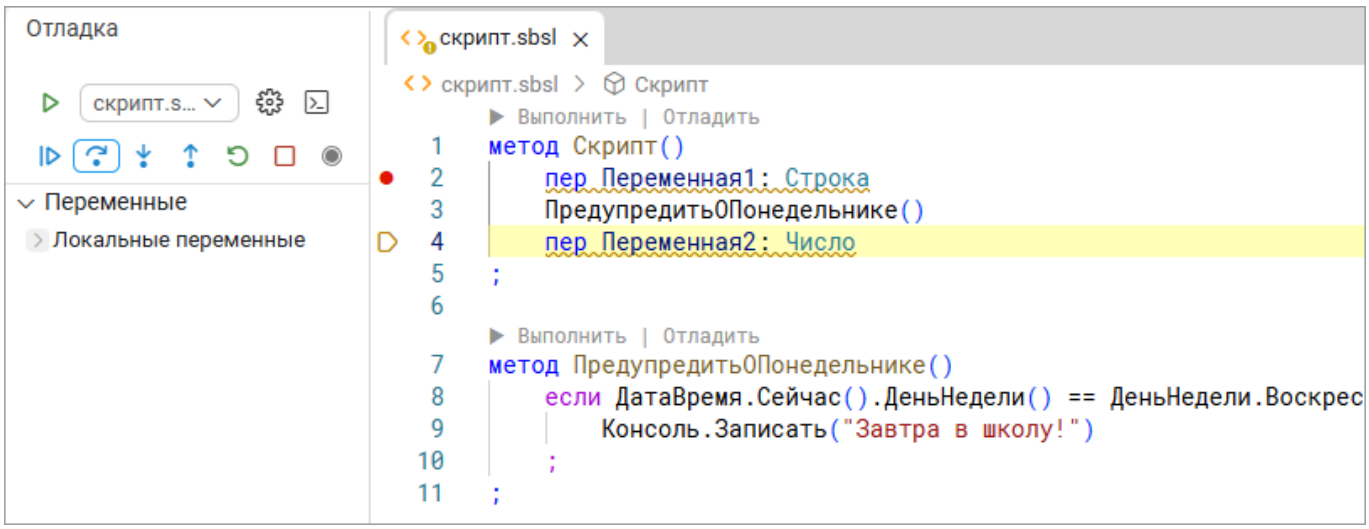
Бывают случаи, когда не нужно заходить внутрь метода, а нужно, чтобы он просто выполнился, без остановки на каждой инструкции.

Например, вы отлаживаете свой скрипт и заранее точно знаете, что внутри метода **ПредупредитьОПонедельнике()** все работает правильно и нет никакой необходимости заходить внутрь него.



Выполните команду **Шаг с обходом** — она расположена в представлении **Отладка** в командной панели, вторая по счёту. Также вы можете вызвать эту команду клавишей **F10** или нажав **Главное меню > Запустить > Шаг с обходом**.

В результате выполнения этой команды метод будет выполнен без остановки, а остановка произойдёт перед исполнением следующей инструкции.



Глава 5. Коллекции, структура, перечисление

Экземпляры, свойства, методы и конструкторы

Экземпляры

Настало время подробнее взглянуть на те значения, с которыми вы имеете дело.

Как вы помните, любое значение относится к тому или иному типу. То есть тип представляет собой множество допустимых значений. Например, тип **СамокатЭлектрический** представляет собой множество всех электросамокатов, которые существуют.

Значением типа **СамокатЭлектрический** является один конкретный самокат. Конкретный самокат — сложное устройство. Он имеет много разных характеристик (цвет, вес, максимальная скорость) и много разных способностей (ускориться, затормозить, подать сигнал, озвучить предупреждение). Важно то, что всеми этими характеристиками и всеми этими способностями обладают все самокаты, все значения этого типа.

Такие сложные значения в «1С:Элементе» называют *экземплярами*. То есть конкретный самокат, который может быть сейчас стоит на стоянке возле остановки, это экземпляр типа **СамокатЭлектрический**. А другой самокат, стоящий рядом с ним — это ещё один экземпляр этого типа.

Как бы вам ни показалось странным, те значения, которые вы уже использовали в своём скрипте: число **17**, значение **2025-06-13T13:15:50.245** типа `ДатаВремя` — тоже являются экземплярами своих типов: типа `Число` и типа `ДатаВремя`. Вообще в «1С:Элементе» все значения «сложные», все они являются экземплярами.

Свойства и методы

Экземпляры отличаются тем, что у них есть свойства, методы и конструкторы.

Если вернуться к самокату, то его цвет, вес, максимальная скорость — это свойства: **Цвет**, **Вес**, **МаксимальнаяСкорость**. Способность ускориться, затормозить, издать сигнал — это методы: **Ускориться()**, **Затормозить()**, **ПодатьСигнал()**.

Таким образом, *свойство* — это некоторая характеристика экземпляра, а *метод* — это некоторое действие, которое можно выполнить над экземпляром или с его помощью.

Дальше, когда речь будет идти о свойстве или о методе какого-то типа, он будет обозначаться через точку от имени типа, по такому принципу:

```
СамокатЭлектрический.МаксимальнаяСкорость
СамокатЭлектрический.Затормозить()
```

Вы уже имели дело со свойствами и с методами некоторых типов.

Например, в самом начале вы написали **Консоль.Записать("Привет")**. Здесь вы использовали метод `Консоль.Записать()`.

Затем вы написали **ДатаВремя.Сейчас()**, чтобы узнать текущее время. Потом вы учились получать начало и конец дня здесь *(49)* тоже с помощью методов типа `ДатаВремя`.

Операция «.» — обращение к свойствам и методам

Обращение к свойствам

Для доступа к свойствам экземпляра используется операция обращения через точку. Она обозначается точкой «.». Точка отделяет имя переменной от имени свойства. Такое обращение можно использовать как для чтения, так и для установки значения свойства.

Можно обращаться к свойствам последовательно. Например, так: **Сейчас.Время.Час**.

```
метод  Скрипт()
      пер  Сейчас  =  ДатаВремя.Сейчас()
      пер  КоторыйЧас  =  Сейчас.Время.Час
;
```

Этот пример «не настоящий», он только для иллюстрации. На самом деле, если вам понадобится получить час от `ДатаВремя`, это можно сделать короче: **Сейчас.Час**. Но можно и так, как в примере:

- Первая точка **Сейчас.Время** из экземпляра `ДатаВремя` получает экземпляр `Дата`;
- Вторая точка **Время.Час** из экземпляра `Дата` получает экземпляр `Число`, обозначающий текущий час.

Обращение к методам

Для работы с методами экземпляров справедливо всё, сказанное выше про свойства, за одним исключением. Свойству экземпляра вы можете присвоить значение (если это позволено), а вот вызов метода в любом случае не может располагаться в левой части инструкции присваивания.

Конструктор

Конструктор — это метод, который позволяет создать экземпляр типа. Например, вам нужен самокат. Вы можете воспользоваться «литералом» самоката — подойти к стоянке самокатов и выбрать один из них. Вот они, вы их видите вживую, вы понимаете, как они выглядят.

Либо вы можете воспользоваться «конструктором» самоката — попросить своего друга дать его самокат покататься. Вы даже не знаете, что за самокат у него, просто он говорил, что самокат хороший, вам этого достаточно. То есть на основе некоторой информации, что у друга есть самокат, вы с помощью «конструктора» получаете экземпляр самоката.

У типа `Число`, например, тоже есть конструктор. С его помощью можно создать число из того, что числом не является. Например, ваш скрипт прочитал текстовый файл с оценками за год. Эти оценки вам нужно использовать в скрипте, но все они представлены в виде строк, так как файл текстовый. В этом случае из строки, обозначающей число, вы с помощью конструктора можете создать само число, экземпляр числа.

С тем, что экземпляр можно создать литералом и конструктором вы уже сталкивались здесь *(44)*, когда работали с типом `ДатаВремя`.

Иерархия типов, тип «Тип»

Тип и контракт

В начале этой книги вы уже знакомились с тем, что такое тип. Но это было «описательное» знакомство, на примерах из окружающего мира (смотрите здесь *(20)*).

Теперь вы знаете уже многие нужные термины и о типах можно поговорить более углублённо.

Тип — это множество допустимых значений (экземпляров) и некоторый набор свойств и методов, присущих этим значениям (экземплярам). В дальнейшем вы будете изучать иерархию типов, поэтому вам понадобится термин «контракт».

Контракт — это и есть набор свойств и методов, присущих типу, а также поведение этих свойств и методов.

Таким образом, можно сказать, что *тип* — это множество допустимых значений и контракт. Это определение вам понадобится дальше.

Иерархия типов

Типы образуют иерархию. Это значит, что в отношении типов можно рассматривать отношение «предок — потомок». В этом случае предка называют *базовым типом*, а потомка — *производным типом*.

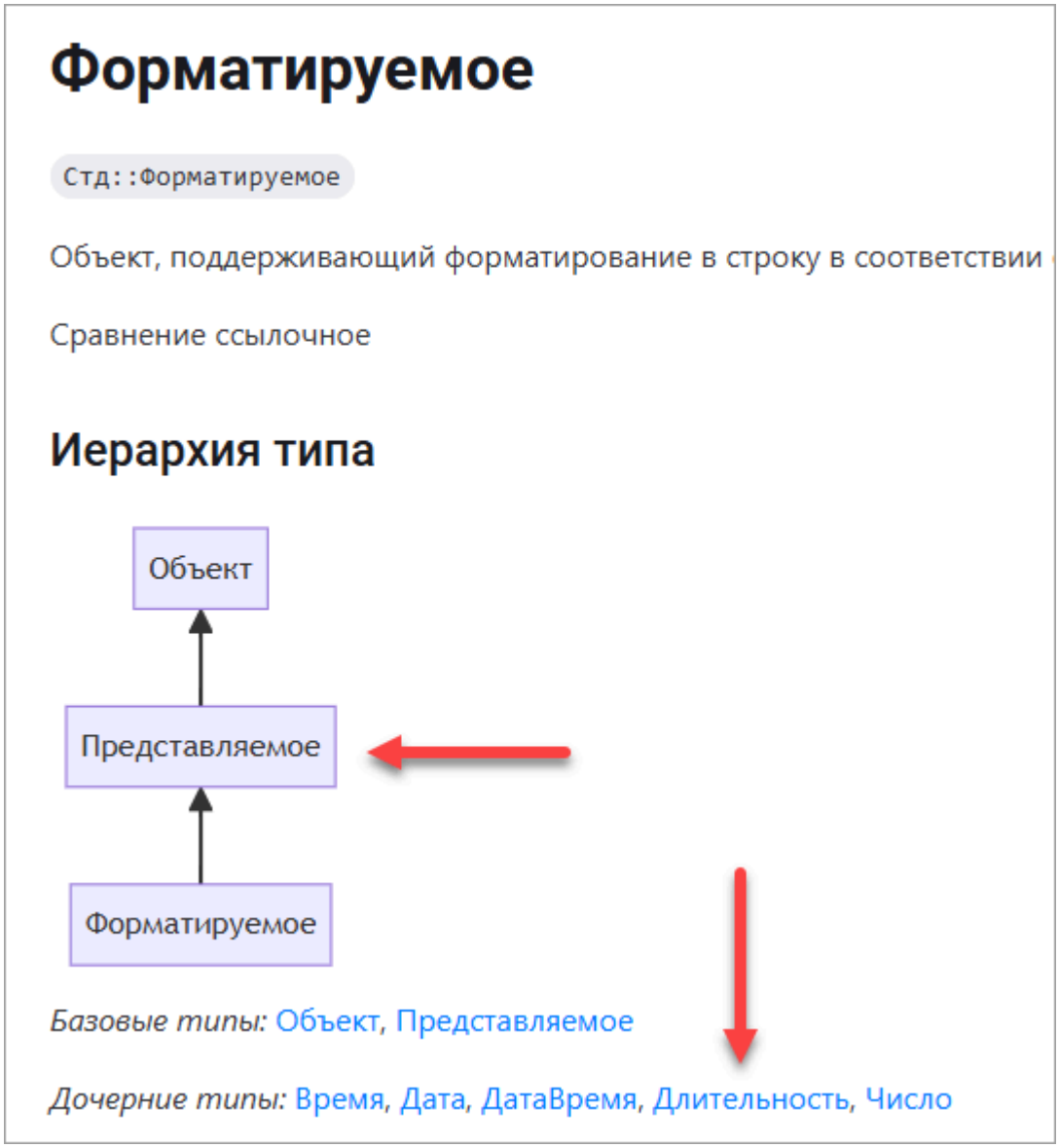
Производный тип наследует контракт своего базового типа. Это значит, например, что если у базового типа есть метод **ВСтроку()**, то у производного типа тоже будет такой метод. Если какой-то тип является потомком типа `Обходимое`, то значения такого типа можно обходить (например, в цикле `для из`). Если тип является потомком типа `Сравнимое`, то значения такого типа могут участвовать в операциях сравнения, и т. д.

Производный тип не просто наследует, но каким-либо образом расширяет либо множество допустимых значений, либо контракт, либо то и другое.

У одного типа может быть несколько базовых типов, то есть один тип может наследовать контракты одновременно у нескольких типов.

В основании всей иерархии типов лежит тип `Объект`. Этот тип является базовым для всех типов, кроме типа `Неопределено`. Про тип `Неопределено` вы узнаете позже *(152)*.

Иерархия типов показана в справке по стандартной библиотеке для каждого типа. Например, тип Форматируемое. Иерархия базовых типов представлена в виде схемы, а дочерние типы перечислены в виде гиперссылок.

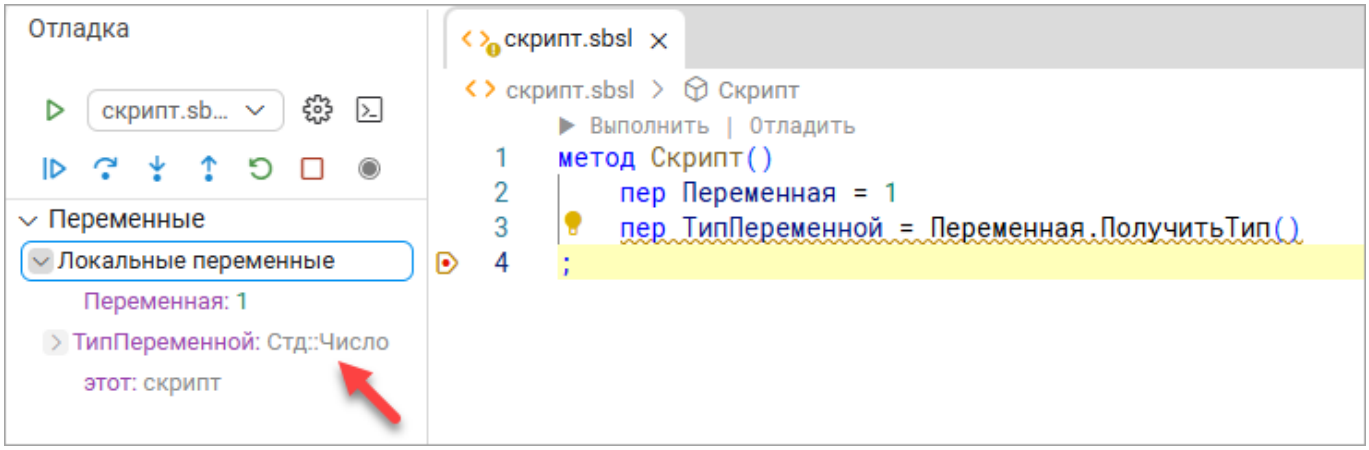


Тип «Тип»

Для определения и сравнения типов, а также для получения информации о иерархии типов в «1С:Элементе» существует специальный тип `Тип`. У каждого из производных типов существует метод `ПолучитьТип()`, который возвращает значение типа `Тип`.

Например, если вы создадите числовую переменную и спросите её тип, то увидите следующее.

```
метод  Скрипт()
    пер  Переменная  =  1
    пер  ТипПеременной  =  Переменная.ПолучитьТип()
;
```



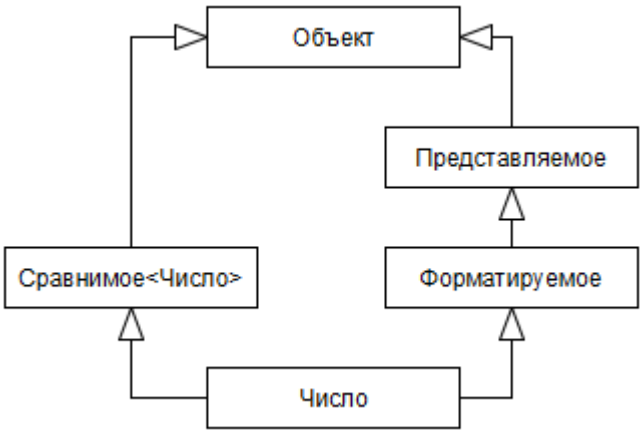
Тип `Тип` имеет литерал, он выглядит как слово **Тип**, после которого в угловых скобках «<» и «>» указывается имя типа.

```
Тип<Строка>            // литерал типа, обозначающего тип Строка
Тип<Число>             // литерал типа, обозначающего тип Число
Тип<Массив<Число>>     // литерал типа, обозначающего тип массива чисел
```

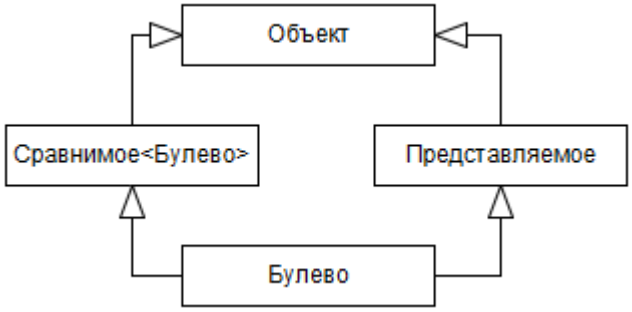
Чтобы получить список базовых типов конкретного типа, используйте свойство `Тип.БазовыеТипы`. Например, вы можете скопировать себе небольшой скрипт, который выведет все базовые типы для указанного. В примере это тип `Строка` (**Тип<Строка>**):

```
метод Скрипт()
    пер БазовыеТипы = СобратьПредков(Тип<Строка>, Ложь)
;

метод СобратьПредков(Значение: Тип, Рекурсия: Булево, Предки: Строка = ""): Строка
    для БазовыйТип из Значение.БазовыеТипы
        если не Рекурсия
            Консоль.Записать("--- " + Значение.Представление())
        ;
        Консоль.Записать(БазовыйТип.Представление())
        СобратьПредков(БазовыйТип, Истина, Предки)
    ;
    возврат Предки
;
```

Например, у типа `Число` список базовых типов будет иметь вид:



У типа `Булево` список базовых типов будет иметь вид:



Присвоение значений

Переменной базового типа можно присвоить значение производного типа, но обратное присваивание невозможно.

Например, числовое значение можно записать в переменную типа **Представляемое**.

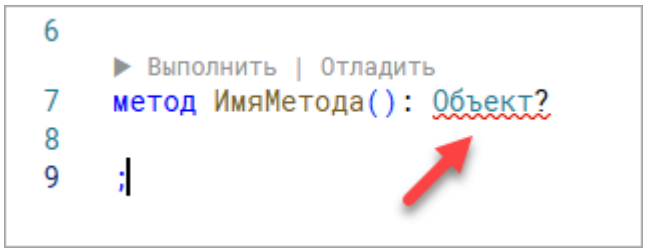
```
метод Скрипт()
    пер Базовая: Представляемое = 5
    пер Производная: Число = 5
    Базовая = Производная
;
```

Однако обратное присвоение невозможно. Если вы попробуете в переменную производного типа поместить значение базового типа, «1С:Элемент» сразу сообщит вам, что это невозможно: **Несовместимые типы: "Представляемое" не может быть присвоен в "Число"**.

```
метод Скрипт()
    пер Базовая: Представляемое = 5
    пер Производная: Число = 5
    Производная = Базовая
;
```

Благодаря тому, что переменные базового типа принимают значения производных типов, переменная типа `Объект` (это тип, лежащий в основании всей иерархии типов) может принять значение любого типа (кроме `Неопределено`). По этой причине, например, в шаблонах синтаксических конструкций, которые

вы вставляете с помощью контекстной подсказки, используется такое описание типа **: Объект?**. То есть в этом месте вы можете написать любой тип либо `Неопределено`. Тип `Неопределено` в этой конструкции обозначается вопросительным знаком.



Операция «это»

Порой возникает необходимость определить, принадлежит значение некоторому типу или нет. Для этого можно использовать операцию `это`. Например:

```
метод Скрипт()
	СравнитьСЧислом("строка 56")
	СравнитьСЧислом(56)
	СравнитьСЧислом(Истина)
;

метод СравнитьСЧислом(Значение: Объект)
	если Значение это Число
		Консоль.Записать(Значение.Представление() + " - это число")
	;
;
```

В результате в терминал будет выведено сообщение **56 - это число**.

Если в проверяемом списке несколько типов, они перечисляются через вертикальную черту «|». Например, **Число|Строка**:

```
метод Скрипт()
	СравнитьСЧислом("строка 56")
	СравнитьСЧислом(56)
	СравнитьСЧислом(Истина)
;

метод СравнитьСЧислом(Значение: Объект)
	если Значение это Число|Строка
		Консоль.Записать(Значение.Представление() + " - это число или строка")
	;
;
```

В результате в терминал будут выведены следующие сообщения.

```
строка 56 - это число или строка
56 - это число или строка
```

> **Примечание:**
>
> Вертикальную черту «|» можно вводить с помощью Alt-ввода *(227)*.

Когда нужно проверить, что переменная не принадлежит перечисленным типам, рекомендуется использовать более понятную операцию проверки с отрицанием `это не`. Например:

```
если Значение это не Строка
```

А не:

```
если не (Значение это Строка)
```

Коллекции и обобщённые типы

Коллекции

Стандартная библиотека содержит несколько типов, называемых *коллекциями*. Это `Массив`, `Соответствие` и `Множество`. Коллекциями их называют потому, что они содержат наборы значений одного или различных типов и позволяют обращаться к этим значениям.

Массив — это упорядоченная коллекция. В ней есть порядок, известно, какой элемент следует за каким. Когда вам понадобится какой-то элемент массива, вы будете брать «первый элемент», «третий элемент» и т. д. Характерные операции над массивом — это перебор элементов с начала или с конца.

В массиве могут присутствовать одинаковые элементы.

Соответствие — это именованная коллекция. В ней нет порядка, но один элемент можно отличить от другого элемента по какому-то признаку — ключу. Например, по цвету. Когда вам понадобится какой-то элемент соответствия, вы будете брать «зелёный элемент», «красный элемент» и т. д. Характерные операции над соответствием — это получение элемента по его признаку.

В соответствии могут присутствовать элементы с одинаковыми значениями, но с разными ключами. Например, один и тот же предмет может быть зелёного цвета, а может быть красного цвета.

Множество — это неупорядоченная неименованная коллекция уникальных элементов. В ней нет порядка и нет никакого признака, по которому можно было бы отличить один элемент от другого. При работе с множествами нет необходимости получать какой-то элемент множества, потому что характерные операции над множествами — это проверка двух множеств на равенство, объединение двух множеств или пересечение множеств.

В множестве не может быть одинаковых элементов.

Обобщённые типы

Обобщённые типы — это типы, которые позволяют типизировать элементы коллекций. То есть благодаря обобщённому типу вы можете сказать, что у вас не просто массив, а массив, состоящий из чисел. У вас не просто множество, а множество, состоящее из строк.

Другими словами, обобщённые типы можно рассматривать как универсальные шаблоны, параметризуемые типами. Есть, например, некий тип **Массив**, который вы можете параметризовать, указав тип его элементов. Получится массив чисел, или массив строк, или массив значений типа **ДатаВремя**. В этом случае типы `Число`, `Строка`, `ДатаВремя` и являются параметрами этого обобщённого типа.

Чем хороши обобщённые типы? Вот пример. Пусть нет обобщённых типов и вы просто говорите, что будете использовать **Массив**. Вы пишете алгоритм, который рассчитан на то, что в массиве будут содержаться только числа.

Но вдруг получается так, что в каком-то месте вашего скрипта вы забыли о том, что с этом массиве должны быть только числа, и передали в алгоритм массив, состоящий из строк.

В этом случае при исполнении скрипта вы получите ошибку, потому что, скорее всего, операции, которые вы выполняли над числами, нельзя выполнять над строками.

Если же у вас есть возможность описать массив с помощью обобщённого типа, указав, что в нём могут содержаться только числа, то всё становится гораздо лучше. Ещё на том этапе, когда вы будете писать код, который помещает в этот массив строки, «1С:Элемент» сообщит вам об ошибке и предупредит, что в этом массиве могут быть только числа.

Описание обобщённого типа

Вы помните, что в «1С:Элементе» всегда нужно указывать тип, кроме тех случаев, когда он понятен из инициализатора переменной. Дальше вы подробно рассмотрите каждую из коллекций, а сейчас просто посмотрите для сравнения, как описываются типы этих коллекций, как выглядит описание обобщённого типа.

Описание обобщённого типа состоит из имени типа и типа его содержимого, указанного в угловых кавычках «<» и «>».

Например, имя типа массива, состоящего из строк, выглядит так: `Массив<Строка>`. Если метод возвращает массив строк, он может быть объявлен так:

```
метод ВернутьМассивСтрок(): Массив<Строка>
    возврат ["первый"]
;
```

> **Примечание:**
>
> Угловые «<» и «>», квадратные «[» и «]» и фигурные скобки «{» «}» можно вводить с помощью Alt-ввода *(227)*.

Например, имя типа соответствия, в котором ключ имеет тип `Строка`, а значение имеет тип `Число`, выглядит так: `Соответствие<Строка, Число>`. Если метод возвращает соответствие из `Строка`: `Число`, он может быть объявлен так:

```
метод ВернутьСоответствиеСтрокаЧисло(): Соответствие<Строка, Число>
    возврат {"рост":180}
;
```

И, наконец, имя типа множества, состоящего из чисел, выглядит так: `Множество<Число>`. Если метод возвращает множество чисел, он может быть объявлен так:

```
метод ВернутьМножествоЧисел(): Множество<Число>
    возврат {1, 3, 5, 7, 9}
;
```

Примечание:

Подробнее про обобщённые типы читайте в руководстве разработчика.

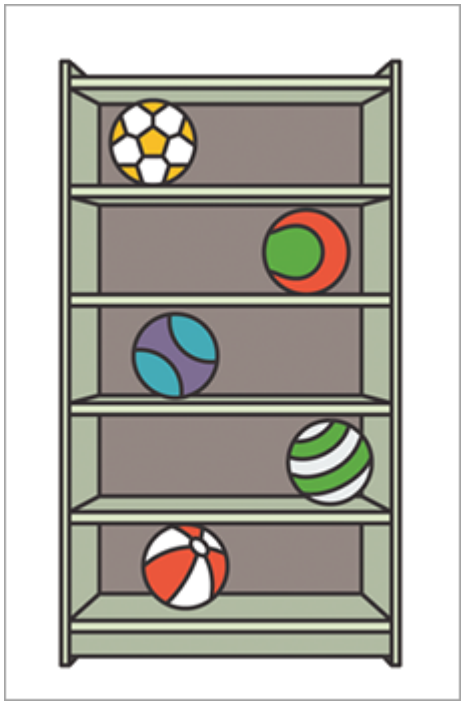
Примечание:

В «1С:Элементе» есть возможность не описывать тип параметра обобщённого типа. Подробнее об этом читайте здесь *(224)*.

Массив

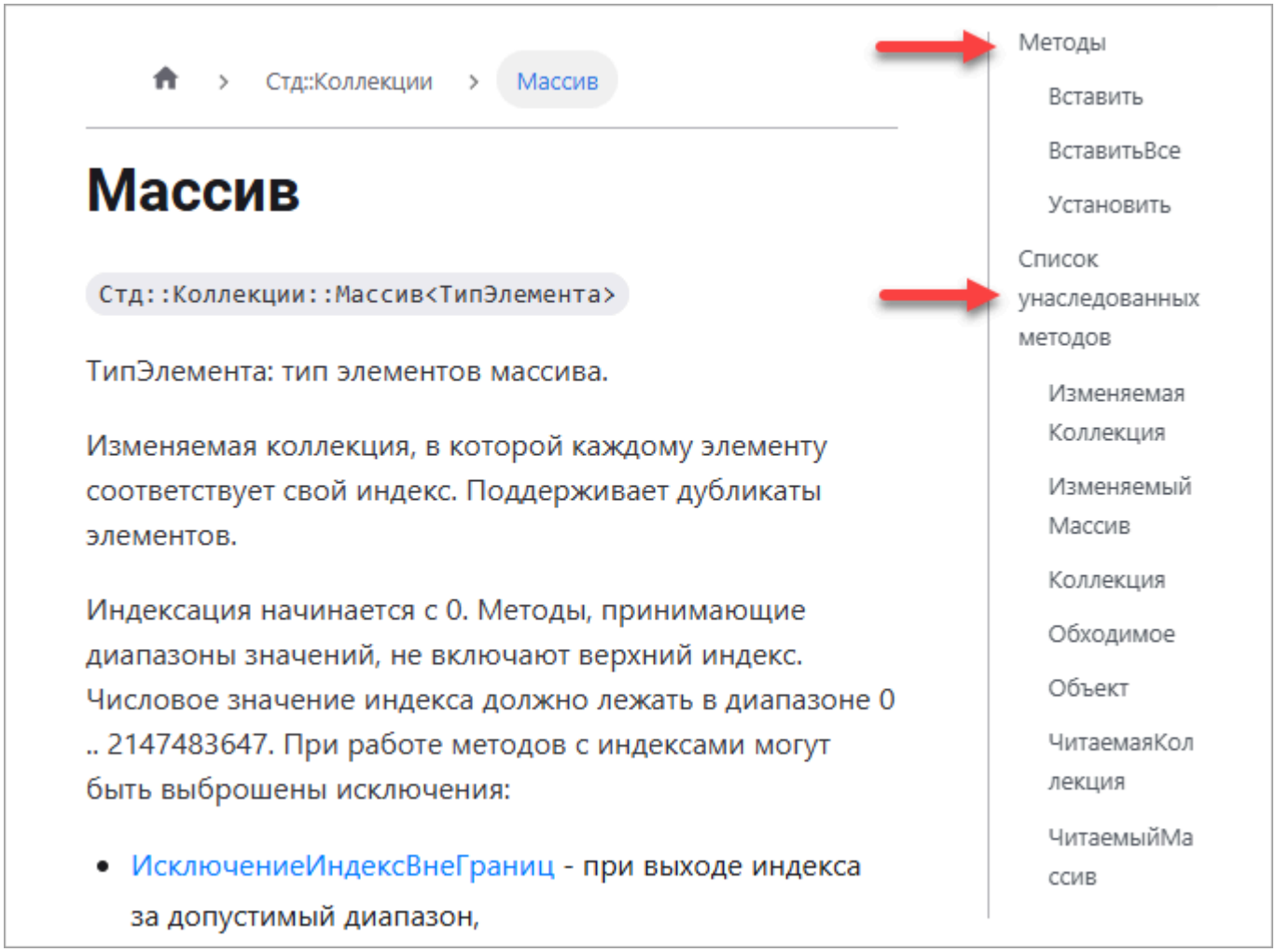
Общая информация

Представьте, что вам нужно где-то хранить мячики. Каждый мячик — это одно значение. Тогда массив — это стеллаж, в котором на каждую полку можно положить только один мячик.



Когда вы захотите взять один из мячиков, вы скажете, например: «Дайте мне мячик со второй полки». Вот так устроен массив — всё очень просто.

В справке по стандартной библиотеке откройте описание типа `Массив`. Если вы забыли, как это делать, посмотрите здесь *(44)*, в разделе про конструктор типа `ДатаВремя`.



На первый взгляд у массива совсем мало методов. Однако посмотрите вниз списка на строку **Список унаследованных методов**. Как вы знаете, типы имеют иерархию, и массив большинство своих методов унаследовал от родителей, от базовых типов.

Если вы нажмёте на **Список унаследованных методов**, то увидите, что методов у массива очень много на самом деле.

Список унаследованных методов

ИзменяемаяКоллекция

Очистить, Удалить, УдалитьВсе, УдалитьКроме

ИзменяемыйМассив

ВставитьНовый, ДобавитьНовый, Перевернуть, Удалить, УдалитьДиапазон, УдалитьПоИндексу

Коллекция

Добавить, ДобавитьВсе

Обходимое

ВМассив, ВСоответствие, ВСоответствиеСКлючами, ВСоответствиеСоЗначениями, ВоМножество, ВсеСоответствуют, ГруппироватьПо, ДляКаждого, ЕдинственныйИлиНеопределено, ЕдинственныйИлиУмолчание, ЕдинственныйИлиУмолчание, ЕстьСоответствия, КакПоследовательность, Максимум, МаксимумПо, Минимум, МинимумПо, НетСоответствий, Объединить, Первый, ПервыйИлиНеопределено, ПервыйИлиУмолчание, ПервыйИлиУмолчание, ПотомСортироватьПо, Преобразовать, ПреобразоватьЛинейно, Пусто, Свернуть, Свернуть, Соединить, Сортировать, Сортировать, СортироватьПо, Среднее, СреднееИлиУмолчание, Сумма, Уникальные, УникальныеПо, Фильтровать, ФильтроватьПоТипу

Объект

ВСтроку, ПолучитьТип, Представление

ЧитаемаяКоллекция

Единственный, Размер, Содержит, СодержитВсе

ЧитаемыйМассив

Граница, Найти, НайтиСКонца, ПодМассив, Получить, Последний, СодержитВсе

Зачем нужен тот или иной метод, вы наверняка поймёте прямо сейчас и даже без объяснений. Не про все методы, но про многие. Например, `Массив.Добавить()` наверняка добавляет один мячик на свободную полку. А `Массив.Удалить()` — удаляет. `Массив.Граница()` — это наверняка для того, чтобы посчитать, сколько мячиков на полках. А `Массив.Очистить()` — освобождает все полки. Так что ничего сложного тут нет.

Литерал и конструктор

Литерал массива может выглядеть следующим образом:

```
пер КраткийЛитерал = [1, 2, 3]
```

Внутри квадратных скобок вы перечисляете значения элементов массива. В данном случае это три числа **1**, **2** и **3**. Поскольку из литерала понятны типы элементов массива, описание типа можно не писать.

Однако может случиться так, что в литерале вы написали числа, но в будущем планируете добавлять в этот массив и строки тоже. В этом случае нужно перечислить перед квадратными скобками «[» «]» внутри угловых скобок «<» «>» все типы, которые могут принимать элементы массива. Типы перечисляются через вертикальную черту «|».

Например, так выглядит литерал массива, который может содержать числа и строки.

```
пер ЛитералСТипом = <Число|Строка>[1, 2, 3]
```

> **Примечание:**
>
> Квадратные «[» «]» и угловые скобки «<» «>», а также вертикальную черту «|» можно вводить с помощью Alt-ввода *(227)*.

С помощью литерала вы можете описать пустой массив. В этом случае указание типа его элементов является обязательным. Например, пустой массив, который может содержать строки.

```
пер ЛитералПустогоМассива = <Строка>[]
```

Конструкторов два. Но это не должно вас смущать. Один конструктор, без аргументов, создаёт пустой массив. Им вы будете пользоваться. Чаще всего используется именно он. Пустой массив строк будет выглядеть следующим образом.

```
пер ПустойМассивСтрока = новый Массив<Строка>()
```

Второй конструктор, с аргументом, создаёт новый массив копированием существующего. Им вы пользоваться в рамках этой книги не будете .

Добавление элементов

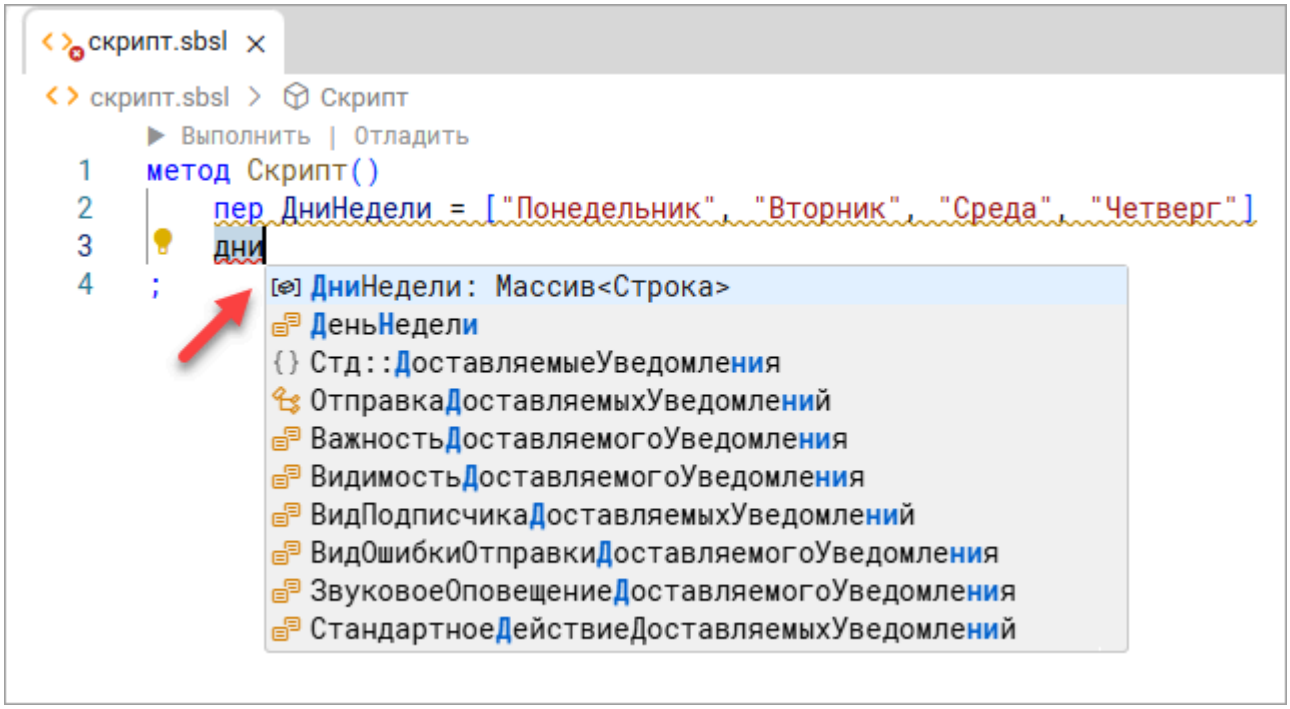
Создайте массив, в котором будут содержаться названия дней недели. Массив будет храниться у вас в переменной **ДниНедели**. Половину массива вы создадите литералом, а другую половину — конструктором.

Сначала с помощью литерала создайте массив из четырёх названий с понедельника по четверг.

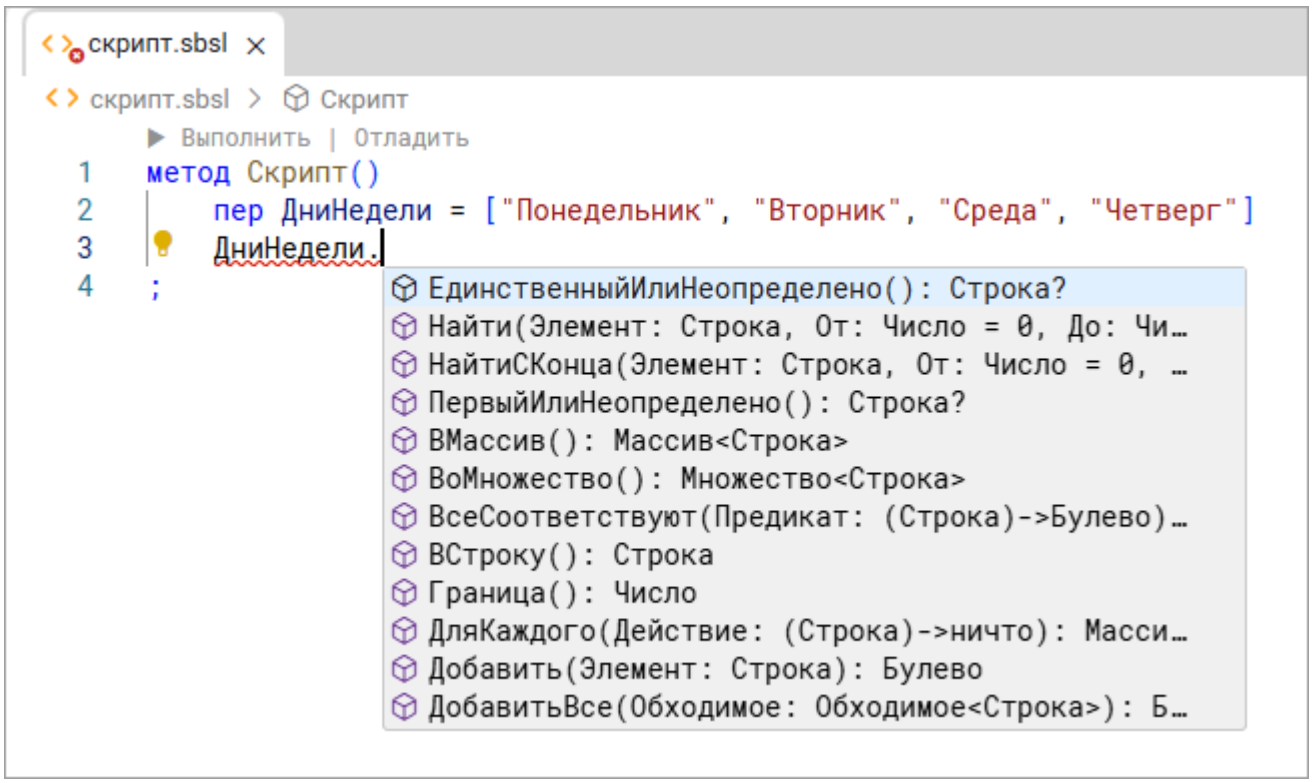
```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг"]
;
```

Три других элемента вы добавите в существующий массив с помощью метода `Массив.Добавить()`. Заодно ещё раз попрактикуетесь пользоваться контекстной подсказкой и копированием строк.

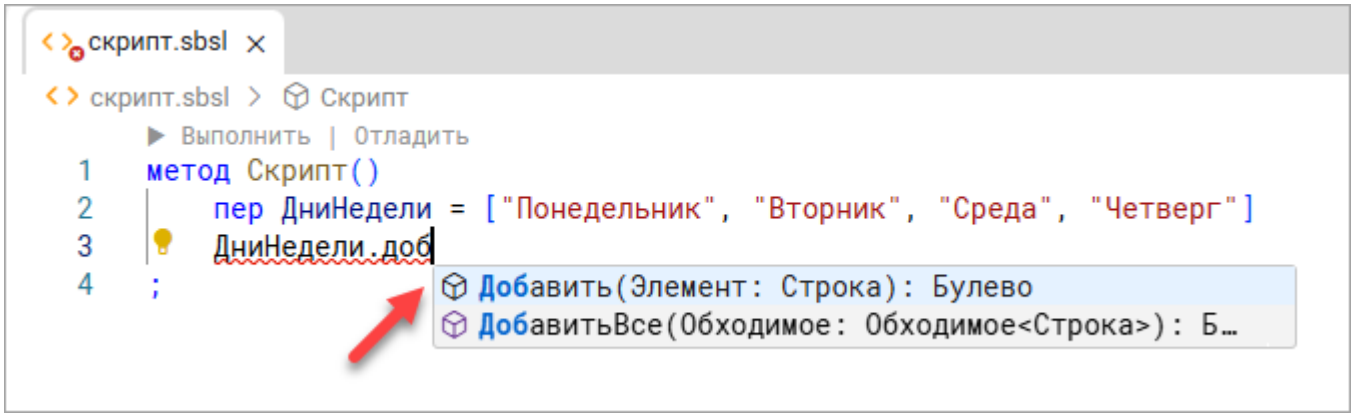
Сначала на следующей строчке напишите **дни**.



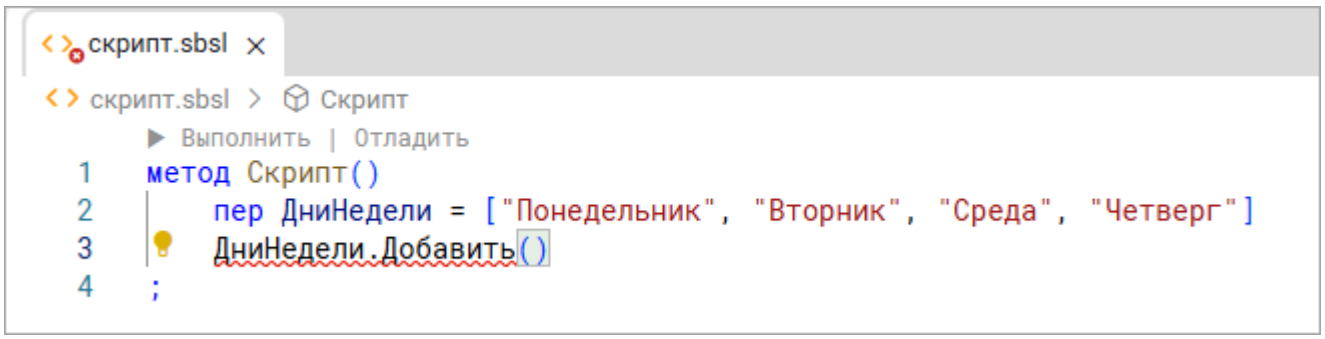
Самый первый элемент — это ваш массив из строк, поэтому нажимайте **Ввод** и ставьте точку. Контекстная подсказка откроется снова.



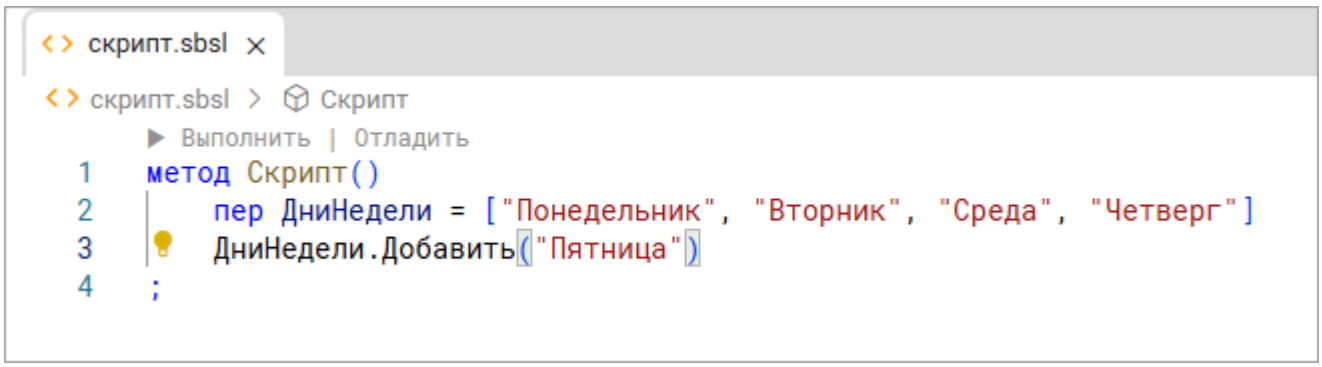
Теперь она показывает вам все свойства и методы, которые есть у экземпляра вашего массива. Вы хотите добавить элемент, поэтому продолжите вводить с клавиатуры **доб**.



Контекстная подсказка покажет, что есть всего два подходящих метода. Первый из них — то, что вам нужно, поэтому нажимайте **Ввод**.

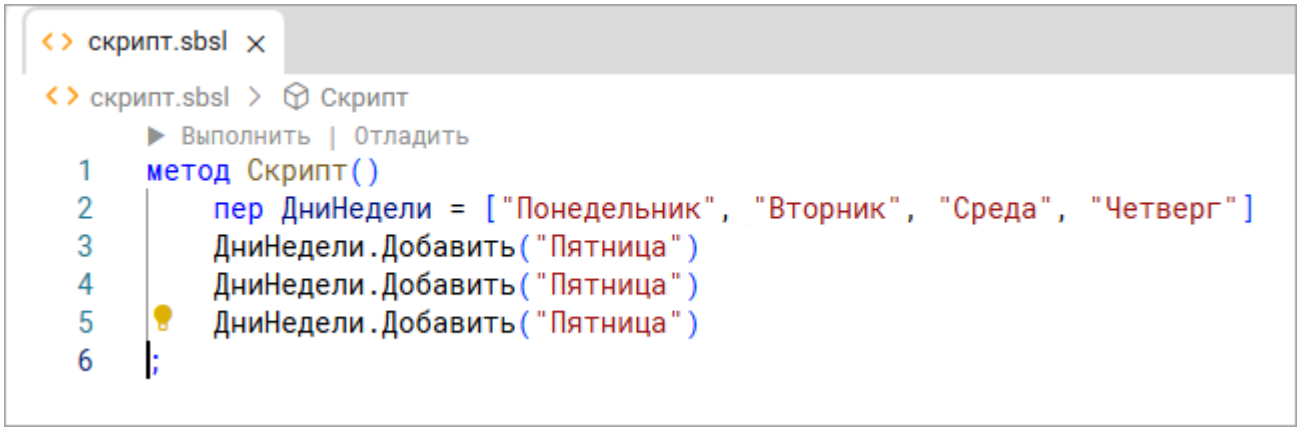


Теперь вам осталось только написать **"Пятница"**.

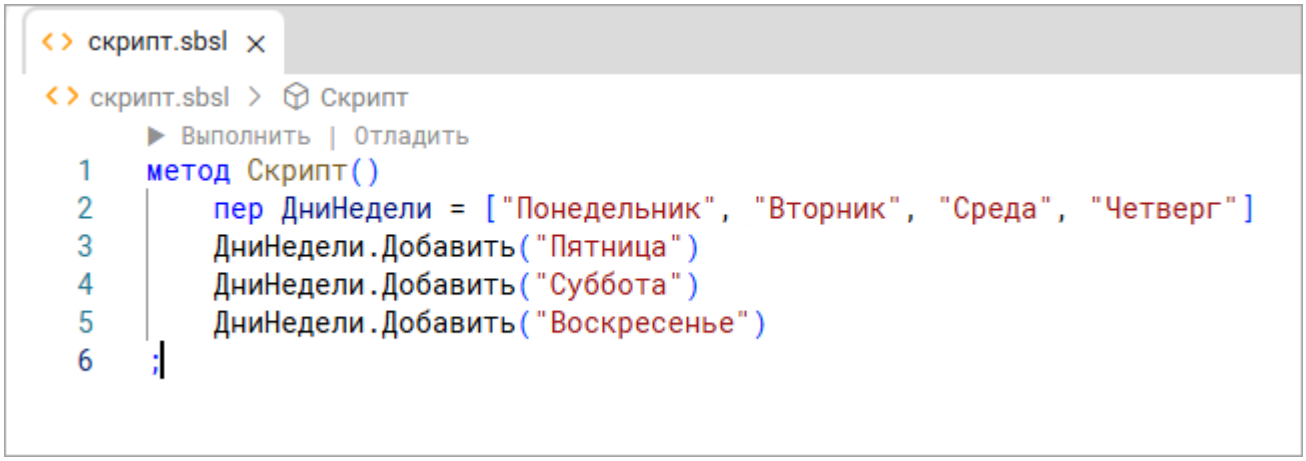


Таким же образом вам осталось добавить в массив ещё два элемента: **"Суббота"** и **"Воскресенье"**.

Для этого вы можете использовать тот же приём *(76)*, который применяли при знакомстве с инструкцией цикла `для по`. Выделите строку, нажав на её номер, а затем нажмите **Ctrl+C** и три раза **Ctrl+V**.



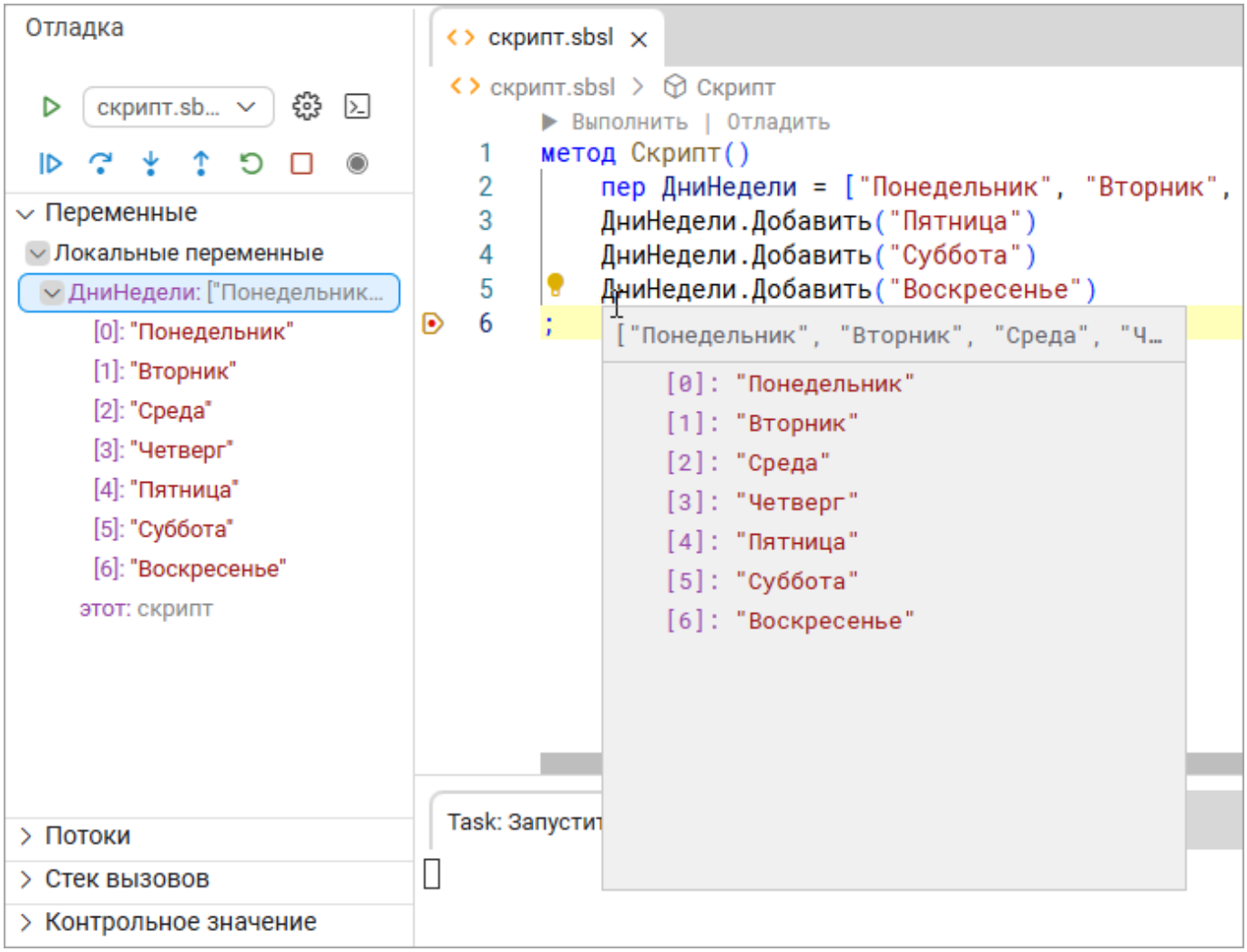
Осталось совсем немного. Дважды щёлкните мышью на втором слове **Пятница** и напишите **Суббота**. Так же исправьте последнее слово **Пятница** на **Воскресенье**.



Всё, массив готов!

Просмотр массива

Теперь возникает вопрос: «Как посмотреть, что находится в массиве?» Установите точку останова на последней строке и запустите скрипт в режиме отладки.



Во-первых, вы можете раскрыть ветку **ДниНедели** в панели **Отладка > Локальные переменные**. Во-вторых, вы можете подвести курсор к имени вашего массива в любом месте скрипта, и «1С:Элемент» откроет вам окно для просмотра этого значения. Каким из способов воспользоваться — дело вашего личного вкуса.

В обоих случаях рядом с именем вашего массива, **ДниНедели**, «1С:Элемент» показывает строку, состоящую из представлений его элементов. Это не всегда удобно, потому что обычно в массиве много элементов.

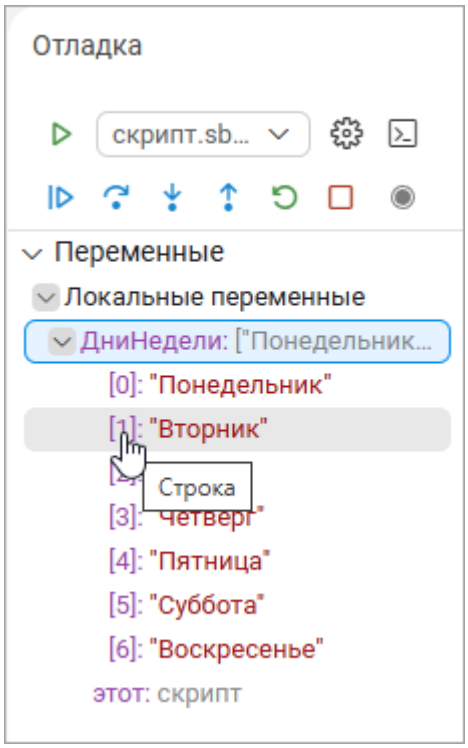
Гораздо удобнее просматривать элементы в виде списка. На каждой строке находится один элемент, они расположены в порядке возрастания своих индексов.

С термином «индекс» вы уже сталкивались, когда изучали строки. Здесь *индекс* имеет тот же самый смысл — это число, порядковый номер элемента в массиве. Индекс первого элемента всегда 0.

> **Примечание:**
>
> Везде в «1С:Элементе», где вам попадётся слово «индекс», знайте: индекс первого элемента всегда равен нулю, а индекс последнего элемента вычисляется по формуле «количество элементов минус единица».

Каждая строка, описывающая элемент массива, начинается с его индекса, а в конце показывается представление этого элемента.

Если вы подведёте курсор к индексу элемента (не важно, в каком из окон), «1С:Элемент» покажет вам тип этого элемента. В вашем случае — `Строка`.

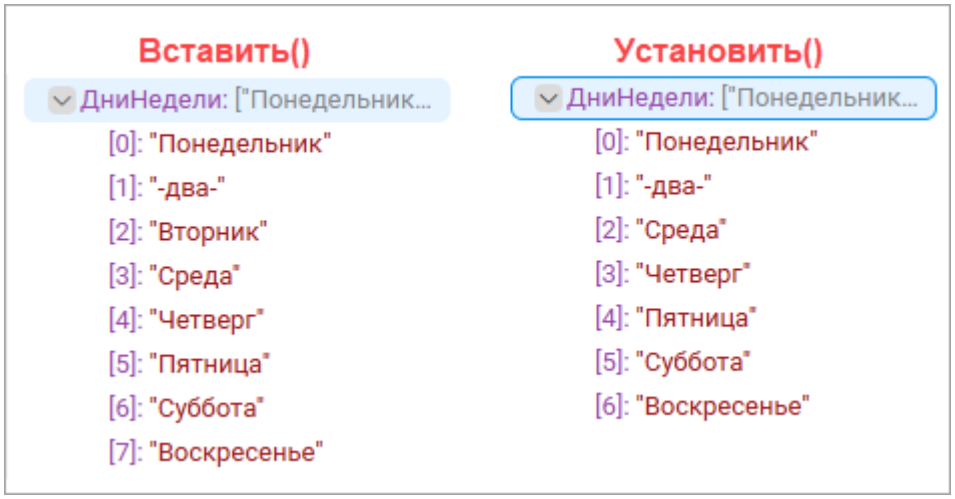


Методы массива

Метод `Массив.Добавить()` вы уже знаете. Он добавляет новое значение в конец массива.

Методы `Массив.Вставить()` и `Массив.Установить()` тоже добавляют одно значение. Не в конец, а в ту позицию, которую вы укажете. При этом `Массив.Вставить()` раздвигает элементы, а `Массив.Установить()` замещает тот элемент, который уже есть в этой позиции.

Например, если вы захотите вставить или установить строку **"-два-"** во вторую позицию массива, то результат будет такой.



Метод `Массив.Граница()` возвращает индекс последнего элемента в массиве. Если вы хотите узнать количество элементов в массиве, то их будет **ДниНедели.Граница() + 1**.

Метод `Массив.Очистить()` удаляет все элементы, а метод `Массив.Удалить()` удаляет только один элемент.

Метод `Массив.Получить()` возвращает значение, которое находится по указанному индексу. А метод `Массив.Найти()` возвращает индекс того значения, которое вы ищете.

То есть **ДниНедели.Найти("Вторник ")** вернёт вам число **1**, а **ДниНедели.Получить(1)** вернёт вам строку **"Вторник"**. Можете попробовать.

31. Задание простое

Создайте массив и запишите в него названия всех месяцев по порядку. Решение смотрите здесь *(298)*.

32. Задание простое

Посмотрите содержимое массива из предыдущего задания в конфигураторе. Решение смотрите здесь *(299)*.

33. Задание простое

В массив из задания 3.33 добавьте два элемента. Один элемент, «--- Начало лета», добавьте перед месяцем, который называется «Июнь». Этот месяц нужно найти по названию. Второй элемент, «--- Конец лета», добавьте после месяца, который называется «Август». Этот месяц тоже нужно найти по названию. Посмотрите в конфигураторе, правильно ли вы выделили летние месяцы. Решение смотрите здесь *(299)*.

34. Задание сложное

С помощью цикла обойдите все элементы массива и добавьте в конец каждого названия текущий год. Чтобы, например, вместо «Январь» стало «Январь 2016 г.». Посмотрите в конфигураторе, правильно ли выглядит результат. Решение смотрите здесь *(300)*.

Сравнение массивов

Массивы можно сравнивать с помощью операции сравнения на равенство `==`. Два массива являются равными, если:

- равны их размеры;
- каждый элемент первого массива содержится во втором массиве;
- порядок следования элементов совпадает.

Следующие массивы равны.

```
["один", 22, 333]
["один", 22, 333]
```

Следующие массивы не равны.

```
["один", 22, 333]
["один", 333, 22]
```

Обрабатывайте ошибочные ситуации

Некоторые методы объектов не всегда могут выполняться успешно. Пример этого вы можете увидеть уже в массиве. Например, если вы посмотрите описание метода `Массив.Найти()` (в справке по стандартной библиотеке), то увидите, что он возвращает `Число` или `Неопределено` — **Число?**.

```
Найти(Элемент: ТипЭлемента, От: Число = 0, До: Число): Число?
```

Число он возвращает тогда, когда выполняется успешно и в массиве есть то, что вы ищете. Но вполне может быть такая ситуация, что в массиве не окажется того, что вы ищете. И тогда он вернёт значение `Неопределено`.

> **Примечание:**
>
> Про тип `Неопределено` вы узнаете позже *(153)*.

Начинающие программисты забывают об этой особенности. Это естественно. Когда вы пишете скрипт, вы точно знаете, что у вас будет в массиве, а чего не будет. Но дело в том, что, когда скрипт начинает работать, он может, например, получить такие данные, которые вы не предусматривали. В результате в массиве не окажется того значения, которое, по вашему мнению, там обязательно должно быть.

Правилом хорошего тона в программировании является обработка не только успешных действий, но и ошибочных ситуаций, которые могут возникнуть.

В данном случае в том месте, где вы бы написали просто **Найти()**, лучше обработать неудачную ситуацию, которая может возникнуть. Например, вы ищете в массиве элемент **Вторник**, а его там нет.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье"]
    пер ВторойДеньНедели = ДниНедели.Найти("Вторник")
    если ВторойДеньНедели == Неопределено
        Консоль.Записать("Вторник не найден!")
        возврат
    ;

    // Дальнейшая работа алгоритма.
    ДниНедели.Установить(ВторойДеньНедели, "ВторойДень")
;
```

Для того случая, когда метод возвращает значение `Неопределено`, вы пишете условие `если`. В нём вы обрабатываете ошибочную ситуацию (выводите сообщение в терминал) и прерываете исполнение метода ключевым словом `возврат`.

Раньше вы использовали `возврат` для того, чтобы вернуть значение из метода в то место, откуда этот метод вызывался. Если метод ничего не возвращает, то `возврат` тоже можно использовать. После него не нужно ничего указывать — исполнение метода будет прервано.

Чтобы посмотреть, как этот пример будет работать без условия `если`, закомментируйте эту инструкцию.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье"]
    пер ВторойДеньНедели = ДниНедели.Найти("Вторник")
    /*если ВторойДеньНедели == Неопределено
        Консоль.Записать("Вторник не найден!")
        возврат
    ;*/

    // Дальнейшая работа алгоритма.
    ДниНедели.Установить(ВторойДеньНедели, "ВторойДень")
;
```

В результате во время выполнения скрипта вы получите сообщение в консоли:

```
Неожиданное значение Неопределено
Scripts::скрипт.Скрипт[10]
```

Не спешите, не ленитесь, обращайте внимание на то, какие значения возвращают методы. Если возможна ситуация, когда выполнение метода заканчивается «неудачей», обрабатывайте такие ситуации. Ваше собственное сообщение поможет быстро понять, в чём проблема.

35. Задание простое

Вы ведёте список учеников, которые поедут на экскурсию. Список постоянно меняется, кто-то добавляется, кто-то вычёркивается. Текущий состав списка вы не знаете.

Вам сообщили, что ученик **Захаров** заболел, и его нужно вычеркнуть из списка.

Создайте массив и заполните его фамилиями **Сергеев**, **Дмитриев**, **Захаров**, **Максимов**. Напишите скрипт, который удаляет из списка ученика по фамилии **Захаров**.

Теперь в инструкциях, которые заполняют массив фамилиями, замените фамилию **Захаров** на фамилию **Александров**. Проверьте, что ваш скрипт по-прежнему работает без ошибок. Решение смотрите здесь *(301)*.

Операция «[]» — обращение к элементу по индексу

Вы уже видели, а может быть, даже и пробовали, как получить элемент массива по его индексу. Для этого можно использовать метод `Массив.Получить()`.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер ВторойДень = ДниНедели.Получить(1)
;
```

Есть более простой способ сделать то же самое. Обычно именно этот способ используют при работе не только с массивами, но и с любыми коллекциями, в которых возможен доступ по индексу. В «1С:Элементе» есть операция обращения к элементу по индексу. Она обозначается квадратными скобками «[» и «]», внутри которых пишется индекс. Например:

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер ВторойДень = ДниНедели[1]
;
```

Примечание:

Квадратные «[» «]» скобки можно вводить с помощью Alt-ввода *(227)*.

Две эти инструкции абсолютно идентичны. И в первом, и во втором случае в переменной **ВторойДень** окажется одно и то же значение — **"Вторник"**.

Вторая инструкция короче и читается легче. Сразу после имени переменной, в которой находится коллекция, ставятся квадратные скобки. В них указывается индекс того элемента, который нужно получить.

Попробуйте самостоятельно получить, например, третий и пятый элементы массива с помощью квадратных скобок.

36. Задание простое

Создайте массив и заполните его названиями дней недели по порядку. С помощью операции `[ ]` получите названия третьего и пятого дней недели. Решение смотрите здесь *(301)*.

37. Задание простое

Создайте массив и заполните его названиями дней недели по порядку. С помощью операции `[ ]` к выходным дням допишите **вых.**. Чтобы, например, вместо **Суббота** получилось **Суббота вых.**. Решение смотрите здесь *(302)*.

Инструкция «для из»

Раньше вы уже познакомились с инструкцией `для по`. Она позволяет многократно выполнять некоторый набор инструкций. `для по` часто используется при работе с коллекциями. Например, для того чтобы перебрать все элементы коллекции и выполнить с ними какие-то действия.

Чтобы не усложнять, напишите самый простой пример. У вас есть массив с названиями дней недели. Вам нужно выбрать из него только будние дни и скопировать их в новый массив.

С помощью `для по` это можно выполнить следующим образом.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    пер ТекущийЭлемент: Строка
    для Индекс = 0 по ДниНедели.Граница()
        ТекущийЭлемент = ДниНедели[Индекс]
        выбор ТекущийЭлемент
        когда "Суббота", "Воскресенье"
            продолжить
```

```
            иначе
                БудниеДни.Добавить(ТекущийЭлемент)
            ;
    ;
;
```

Как работает этот пример? Сначала вы создаёте новый массив, в который будете помещать будние дни, и переменную для хранения текущего элемента массива. После этого последовательно, от нуля до последнего индекса, перебираете все элементы массива.

Значение текущего элемента вы получаете по его индексу с помощью операции `[ ]`. Если это значение **"Суббота"** или **"Воскресенье"**, то вы ничего не делаете и переходите к следующей итерации цикла, то есть к его началу. Для этого используется новое для вас ключевое слово `продолжить`.

Во всех других случаях вы добавляете название буднего дня в свой заранее созданный массив.

Скопируйте этот пример в свой скрипт, в режиме отладки посмотрите по шагам, как он работает. В том числе посмотрите, как работает ключевое слово `продолжить`.

Такие задачи, когда нужно перебрать все элементы коллекции, встречаются часто. Поэтому в «1С:Элементе» существует специальная форма инструкции цикла. Она обозначается `для из` — обход коллекции. Тот же пример с её использованием будет выглядеть так.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    для ТекущийЭлемент из ДниНедели
        выбор ТекущийЭлемент
        когда "Суббота", "Воскресенье"
            продолжить
        иначе
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

Вы видите, что выглядит этот пример проще и читается легче.

Если сравнить оба примера, то видно, что по сути инструкция `для из` заменила собой сразу три строки, которые были в старом примере.

```
// Старый пример
    пер ТекущийЭлемент: Строка
    для Индекс = 0 по ДниНедели.Граница()
        ТекущийЭлемент = ДниНедели[Индекс]


// Новый пример
    для ТекущийЭлемент из ДниНедели
```

Инструкцию `для из` используют тогда, когда порядок обхода коллекции не важен. Когда важно лишь перебрать все элементы, которые есть в коллекции.

Попробуйте в своём скрипте, как работает второй вариант этого примера.

В ключевом слове `продолжить`, с которым вы познакомились в этом примере, нет ничего сложного. Оно используется только внутри циклов и только для такой задачи, которая показана в этом примере. То есть когда нужно перейти к очередной итерации цикла, не выполняя операторы, следующие далее.

Кроме `продолжить` внутри цикла может использоваться и другое ключевое слово — `прервать`. Оно тоже очень простое. Оно без всяких условий прекращает исполнение цикла и переходит к инструкции, которая расположена после точки с запятой «;», закрывающей тело цикла.

Ключевое слово `прервать` может понадобиться вам тогда, когда вы точно знаете, что перебирать коллекцию дальше нет смысла. Например, в вашем массиве дни недели всегда расположены в правильном порядке, и вам нужно выбрать все будние дни. Если вы перебираете их с начала и дошли до субботы, то понятно, что дальше их перебирать нет смысла. Следующий за субботой день тоже будет выходной.

38. Задание простое

Используйте такой скрипт.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    для ТекущийЭлемент из ДниНедели
        выбор ТекущийЭлемент
        когда "Суббота", "Воскресенье"
            продолжить
        иначе
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

Представьте, что для будних дней вам нужно не просто выполнить одну строку **БудниеДни.Добавить(ТекущийЭлемент)**, а реализовать в этом месте большой и сложный алгоритм. Прятать такой алгоритм внутрь условия `выбор` нехорошо и неудобно.

В этом случае обычно поступают следующим образом. В начале цикла сразу же определяют, подходит очередной день для выполнения алгоритма или нет. Если не подходит (выходной), то просто переходят к следующей итерации цикла. Больше никаких проверок не делают. В результате получается, что инструкции, написанные после `выбор`, будут выполняться только для подходящих элементов.

Измените показанный скрипт так, чтобы он соответствовал этой стратегии. С помощью пошаговой отладки посмотрите, как работает цикл. Решение смотрите здесь *(302)*.

39. Задание простое

Используйте такой скрипт.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    для ТекущийЭлемент из ДниНедели
        если ТекущийЭлемент == "Суббота" или ТекущийЭлемент == "Воскресенье"
            продолжить
        иначе
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

Обратная ситуация. Алгоритм, который нужно выполнить для будних дней, небольшой и несложный.

В этом случае с помощью инструкции `выбор` отбирают только те элементы, которые подходят для этого алгоритма, и выполняют алгоритм. Получается, что после `выбор` нет никаких инструкций и для неподходящих элементов ничего не выполняется.

Измените показанный скрипт так, чтобы он соответствовал этой стратегии. С помощью пошаговой отладки посмотрите, как работает цикл. Решение смотрите здесь *(302)*.

40. Задание простое

Используйте такой скрипт.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    пер ТекущийЭлемент: Строка
    для Индекс = 0 по ДниНедели.Граница()
        ТекущийЭлемент = ДниНедели[Индекс]
        выбор ТекущийЭлемент
        когда "Суббота", "Воскресенье"
            продолжить
        иначе
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

Дни недели расположены в массиве в правильном порядке. Вы обходите их в цикле в том же правильном порядке. Вы точно знаете, что после субботы будет воскресенье, которое вам не нужно.

Измените скрипт так, чтобы не анализировались те дни, которые вам заведомо не нужны. С помощью пошаговой отладки посмотрите, как работает цикл. Решение смотрите здесь *(303)*.

41. Задание сложное

Создайте массив и в цикле заполните его годами с 2000 по 2030. Затем из этого массива отберите только високосные годы и поместите их в другой массив. Високосным считается год, который делится на 4 без остатка.

Объясните, почему вы использовали ту или иную инструкцию цикла. Посмотрите в отладке, сколько високосных лет у вас получилось. Решение смотрите здесь *(303)*.

Удаляйте элементы с конца

Инструкция `для из` очень удобная и простая. Так и хочется ею пользоваться! Однако есть ситуация, когда она не подходит. Эта ситуация — удаление элементов массива в процессе обхода.

Во-первых, инструкция `для из` не допускает изменения коллекции в процессе своего выполнения. Во-вторых, для правильного удаления элементов массива нужно обходить массив в обратном порядке, поэтому нужно пользоваться либо инструкцией `для по`, либо ещё одной инструкцией цикла `пока`. Разберите эту ситуацию подробно.

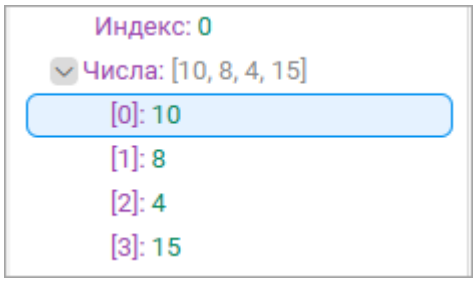
Чтобы понять проблему, скопируйте себе следующий скрипт.

```
метод Тест1()
    пер Числа = [10, 8, 4, 15]
    для Индекс = 0 по Числа.Граница()
        если Числа[Индекс] < 10
            Числа.УдалитьПоИндексу(Индекс)
        ;
    ;
;
```

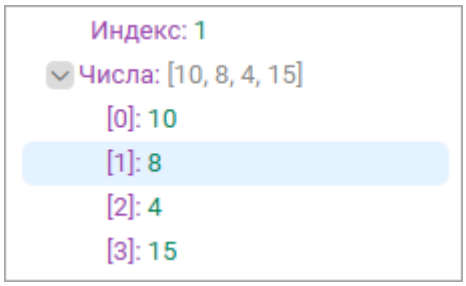
Задача простая. В массиве четыре числа. Нужно удалить из этого массива все числа, которые меньше чем 10.

Поставьте точку останова на инструкции `если` и запустите отладку.

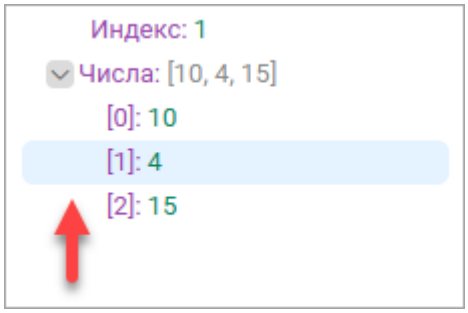
На первой итерации цикла **Индекс** у вас — **0**, а число, которое вы анализируете — **10**.



На второй итерации цикла **Индекс** у вас — **1**, а число, которое вы анализируете — **8**.

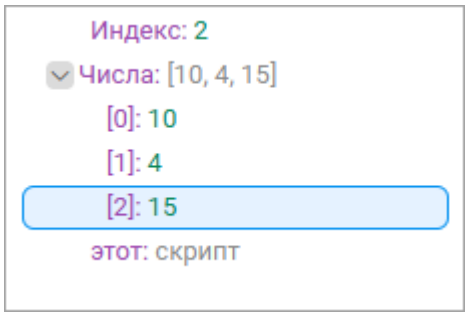


И тут вы понимаете, что этот элемент нужно удалить. Но коллекции не терпят пустых мест. Как только вы его удалите, остальные элементы коллекции переместятся, чтобы занять освободившееся место.



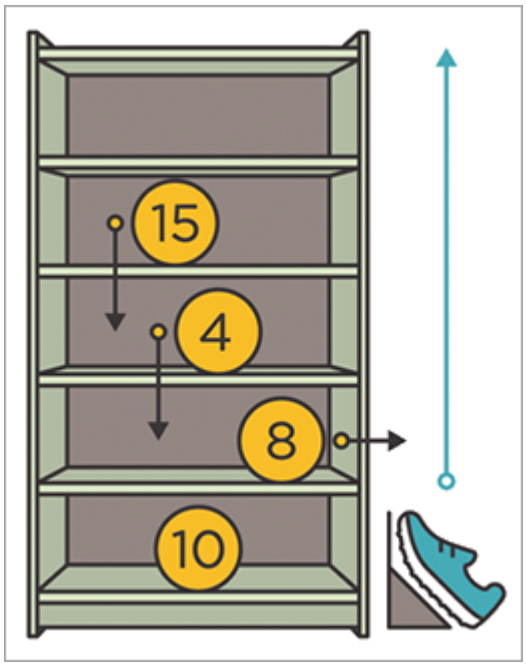
Получается, что на втором месте оказалось число **4**, которое вы ещё не обрабатывали.

Но цикл продолжается, и вы, естественно, переходите к следующему элементу коллекции.



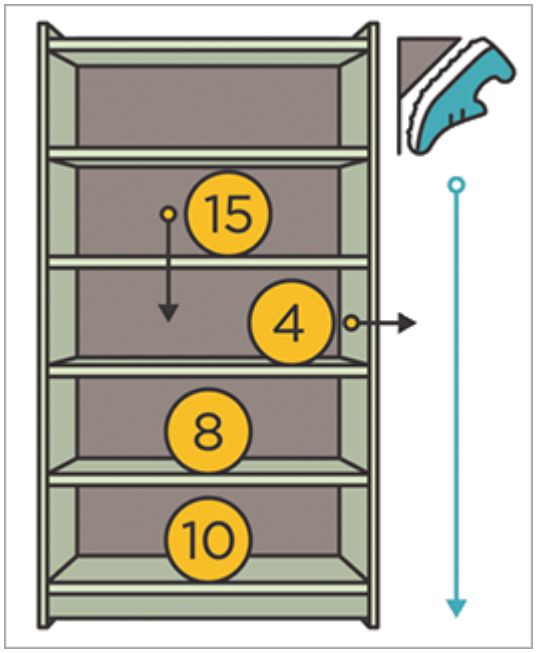
Теперь ваше текущее число **15**. А **4** «проскочило» мимо вас и осталось в массиве.

Если вспомнить, что вы представляли массив как стеллаж, то, когда вы обходите его в прямом порядке и удаляете один из мячиков, на освободившееся место падает мячик, который вы ещё не обрабатывали, который вы должны были обработать только на следующем шаге.



Но вы двигаетесь дальше, и необработанный мячик проскакивает мимо вас.

Чтобы такого не случалось, нужно двигаться в обратную сторону, то есть от конца к началу.



Тогда, если вы удалите элемент **4**, вниз «провалится» элемент с той полки, которую вы уже обрабатывали, — **15**. И это не страшно.

Как это сделать в «1С:Элементе»? Несложно. Есть два способа.

Инструкция `для по` *в обратном направлении*

Направление обхода можно поменять прямо в инструкции `для по`. Для этого надо поменять местами начальное и конечное значения итератора и добавить ключевое слово `вниз`.

```
метод Скрипт()
    пер Числа = [10, 8, 4, 15]
    для Индекс = Числа.Граница() вниз по 0
        если Числа[Индекс] < 10
            Числа.УдалитьПоИндексу(Индекс)
        ;
    ;
;
```

Инструкция `пока`

Эта инструкция предоставляет вам полную свободу в организации цикла. Единственное, что вы можете указать, — это условие выхода, которое будет проверяться на каждой итерации цикла. Все остальное: какая переменная цикла у вас будет, как она будет меняться, с каким шагом, в какую сторону, — вы описываете самостоятельно.

Более того, даже условие выхода из цикла можно не использовать, а написать просто **пока Истина**, и цикл будет «крутиться» до тех пор, пока в теле цикла не выполнится `прервать`.

Возвращаясь к задаче обхода массива в обратном направлении, можно написать такой скрипт.

```
метод Тест3()
    пер Числа = [10, 8, 4, 15]
    пер ТекущийИндекс = Числа.Граница()
```

```
    пока ТекущийИндекс >= 0
        если Числа[ТекущийИндекс] < 10
            Числа.УдалитьПоИндексу(ТекущийИндекс)
        ;
        ТекущийИндекс -= 1
    ;
;
```

Чтобы обходить цикл, вы создаёте переменную **ТекущийИндекс**. В ней будет находиться индекс того элемента массива, который нужно обработать. Сначала вы помещаете в эту переменную индекс последнего элемента — **Числа.Граница()**.

В конце тела цикла вы каждый раз уменьшаете этот индекс на единицу. Это позволяет вам двигаться от конца к началу — **ТекущийИндекс -= 1**.

В условии цикла вы проверяете, не стал ли текущий индекс отрицательным — **пока ТекущийИндекс >= 0**. Как только он становится меньше нуля, исполнение цикла завершается.

Таким образом вы обходите все индексы от последнего к нулевому.

Скопируйте этот пример в свой скрипт и посмотрите по шагам как он работает.

Многомерные массивы

До сих пор вы рассматривали одномерные массивы. Одномерный массив — это линейный список, очередь в кассу, например. Человек за человеком.

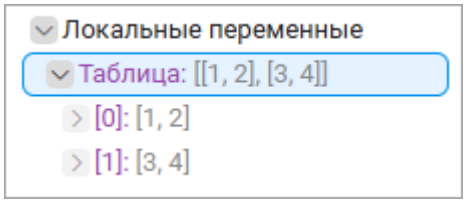
Однако у массива может быть несколько изменений. Например, может быть двумерный массив. Двумерный массив похож на таблицу: есть строки, есть колонки. Двумерный массив —это когда каждый элемент массива в свою очередь тоже является массивом. Например, строки таблицы — это массив. Но каждая строка, в свою очередь, состоит из нескольких колонок, которые тоже являются массивом.

Массивы могут быть трёхмерными, четырёхмерными и т. д.

Для знакомства скопируйте в свой скрипт следующий двумерный массив.

```
метод Скрипт()
    пер Таблица = [[1, 2], [3, 4]]
    Таблица[0][0] = 8
;
```

Остановитесь в отладке на третьей строке и посмотрите, как выглядит **Таблица**.



Она действительно похожа на таблицу: первый массив — это строки, вторые массивы — колонки.

Если вы сделаете шаг, то поменяете значение «углового» элемента — первого элемента из массива **[1, 2]**.

Литерал пустого двумерного массива чисел и добавление элемента в него могут выглядеть так.

```
метод  Скрипт()
      пер  ДвумерныйМассив  =  <Массив<Число>>[]
      ДвумерныйМассив.Добавить([35])
;
```

Примечание:

Работа с массивами на углублённом уровне описана здесь *(193)*.

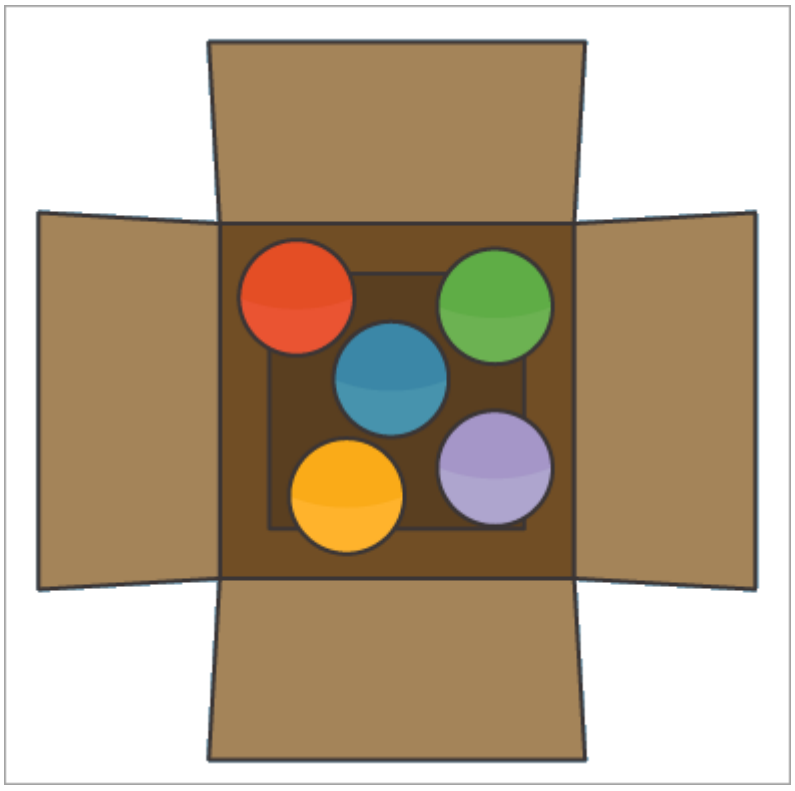
Примечание:

- О работе с массивами читайте в руководстве разработчика.
- Описание типа `Массив` читайте в справке по стандартной библиотеке.

Соответствие

Общая информация

Ещё одна коллекция, с которой вы познакомитесь, называется соответствием. Она, как и массив, может содержать в себе набор некоторых значений. Но только если массив это стеллаж, на котором значения-мячики находятся в порядке, друг за другом, то соответствие похоже на обычную коробку, в которой эти мячики лежат в беспорядке.



Как же тогда отличить один элемент соответствия от другого, если у них нет никакого порядка? Тут кроется особенность, отличающая соответствие от других коллекций.

Вы видели, что элементами массива являются сами хранимые значения. Элементами соответствия являются экземпляры специального «промежуточного» типа, внутри которых уже находятся хранимые значения.

Таким промежуточным типом является тип `КлючИЗначение`. Экземпляр этого типа представляет собой, грубо говоря, коробку из двух отделений. В одно отделение, `Значение`, вы кладёте то значение, которое хотите хранить в соответствии, а в другое отделение этой коробки, `Ключ`, кладёте то значение, которое будет отличать ваше хранимое значение от других значений, находящихся в этом соответствии.

Литерал и конструктор

Литерал соответствия может выглядеть следующим образом:

```
пер КраткийЛитерал = {"Длина":180, "Ширина":90, "Высота":20}
```

Внутри фигурных скобок вы перечисляете экземпляры `КлючИЗначение`. Экземпляры отделяются друг от друга запятой. Каждый из экземпляров записывается в форме ***ключ*** `:` ***значение***.

Таким образом, в примере хранятся числа — например, размеры товара. Его длина, ширина и высота. Строковые значения **"Длина"**, **"Ширина"** и **"Высота"** являются ключами, по которым можно отличить одно хранимое значение от другого.

Если в вашем литерале есть не все типы значений, которые вы собираетесь хранить в соответствии, то в этом случае в литерале нужно указать типы ключей и значений. Они перечисляются перед фигурными скобками «{» «}» внутри угловых скобок «<» «>». Тип ключа и тип значения отделяются друг от друга запятой «,». Если ключ или значение могут принимать значения нескольких типов, то эти типы перечисляются через вертикальную черту «|».

Вот, например, литерал соответствия, в котором значения могут быть как числами, так и строками.

```
пер ЛитералСТипом = <Строка, Число|Строка>{"Длина":180, "Ширина":90, "Высота":20}
```

Такой литерал позволит вам в дальнейшем добавить в это соответствие **"Материал":"Дерево"**.

> **Примечание:**
>
> Фигурные «{» «}» и угловые скобки «<» «>», а также вертикальную черту «|» можно вводить с помощью Alt-ввода *(227)*.

С помощью литерала вы можете описать пустое соответствие. В этом случае указание типа его ключей и значений является обязательным. Например, пустое соответствие со строковыми ключами, в котором хранятся числа.

```
пер ПустоеСоответствиеСтрокаЧисло = <Строка, Число>{:}
```

У соответствия два конструктора. Один создаёт пустое соответствие, другой создаёт соответствие копированием существующего. Пустое соответствие со строковыми ключами, в котором хранятся числа, будет выглядеть следующим образом.

```
пер ПустоеСоответствиеСтрокаЧисло = новый Соответствие<Строка, Число>()
```

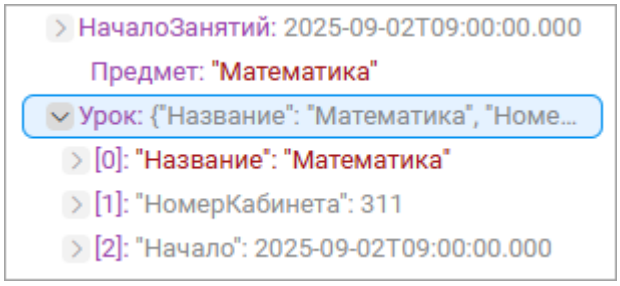
Просмотр соответствия и обращение к элементам

Чтобы познакомиться с соответствием, скопируйте себе следующий скрипт.

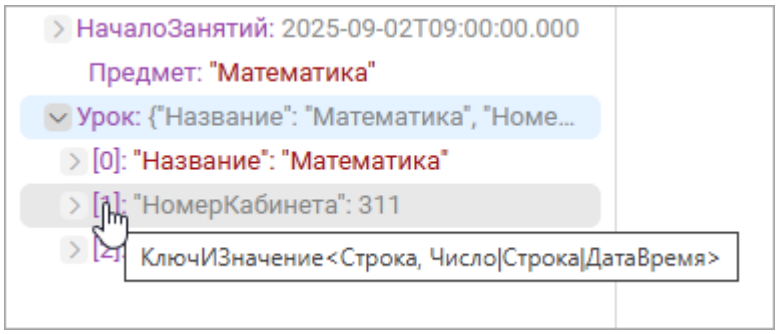
```
метод Скрипт()
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
    пер НачалоЗанятий = Урок["Начало"]
    пер Предмет = Урок["Название"]
;
```

В нём создаётся соответствие, описывающее один урок: название урока, номер кабинета и время начала. Затем выполняется обращение к двум его элементам.

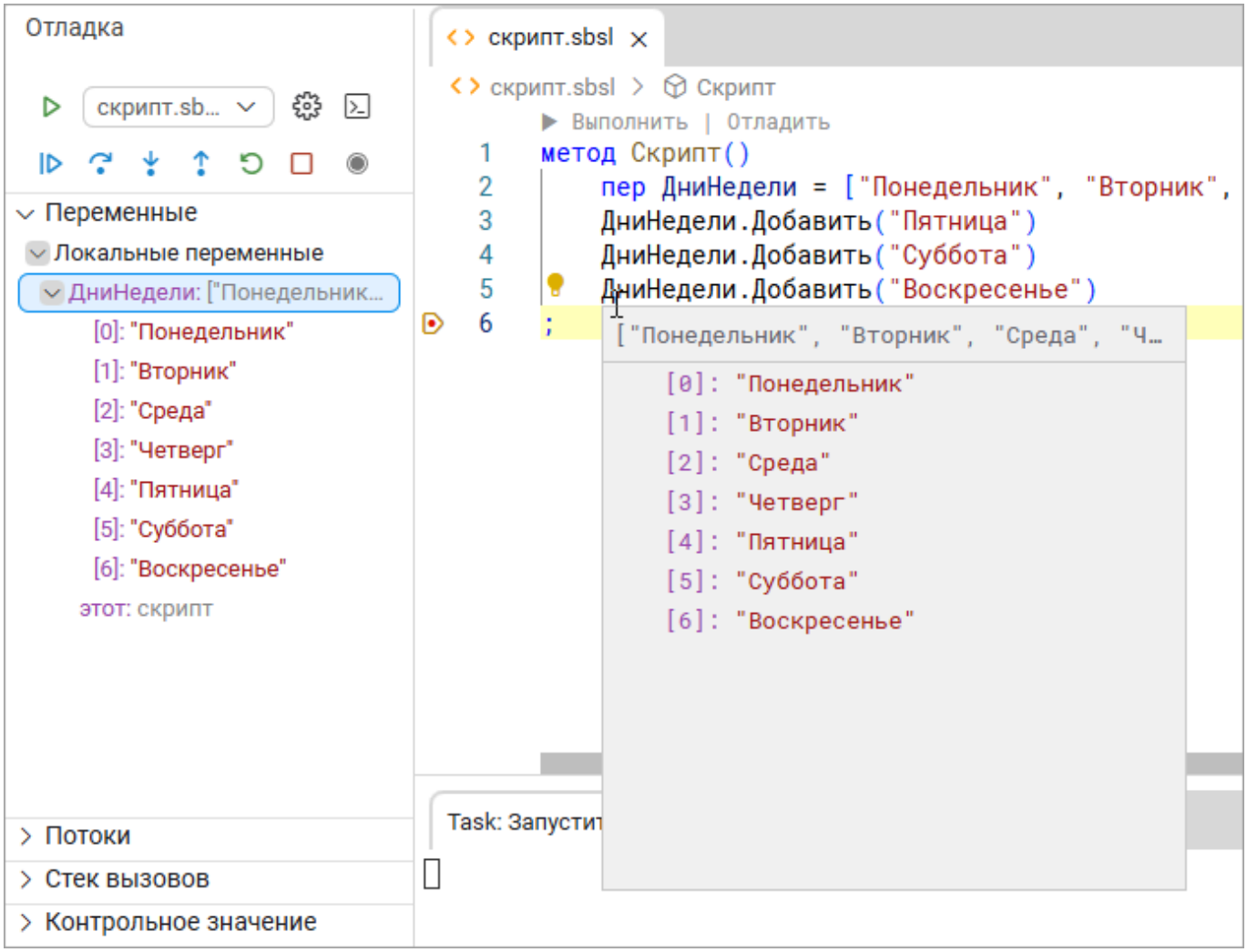
Установите точку останова на последней строке и запустите отладку.



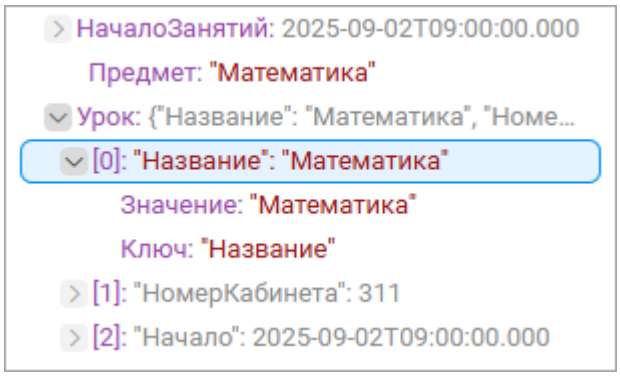
Раскройте строку **Урок**. В ней вы видите «содержимое» соответствия. В соответствии есть три элемента, каждый из которых представлен экземпляром `КлючИЗначение`. Вы можете в этом убедиться, если подведёте курсор к любому из ключей: **Название**, **Начало** или **НомерКабинета**.



Если вы вспомните массив, то он выглядел иначе: элементами были сами значения, а «идентификаторами» этих значений был индекс.



У массива нельзя было «развернуть» строку со значением, а у соответствия есть такая возможность. Если вы развернёте любую строку соответствия, но увидите «части», из которых она состоит: **Ключ** и **Значение**.



Как можно обратиться к элементу соответствия? Как прочитать его значение или изменить его?

В массиве для этого можно было использовать операцию `[ ]`, передавая в неё индекс элемента.

Глядя на внутреннее устройство соответствия, можно подумать, что аналогичный подход будет работать и здесь.

Действительно, чтобы обратиться к элементу соответствия, нужно использовать операцию `[ ]` и указывать в ней ключ элемента:

```
пер НачалоЗанятий = Урок["Начало"]
пер Предмет = Урок["Название"]
```

Изменять значение элемента соответствия можно только таким образом. А вот прочитать хранящееся там значение, можно так, а можно с помощью метода `Соответствие.Получить()`. В него тоже нужно передавать значение ключа.

Методы соответствия

Вставить элемент

Чтобы вставить элемент в соответствие, используйте метод `Соответствие.Вставить()`:

```
метод Скрипт()
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
    Урок.Вставить("Название", "Физика")
    Урок.Вставить("Учитель", "Сергеев")
;
```

В результате соответствие **Урок** будет содержать значение **{"Название":"Физика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}, "Учитель":"Сергеев"}**.

Если в соответствии есть элемент с добавляемым значением ключа, то значение этого элемента заменится (ключ **Название**). В противном случае в соответствие добавляется новый элемент (ключ **Учитель**).

Если вы не хотите случайно изменить существующие значения соответствия, а хотите только добавлять в него значения с новыми ключами, используйте метод `Соответствие.ВставитьЕслиОтсутствует()`. В том случае, если в соответствии уже есть элемент с таким ключом, этот метод ничего не делает.

```
метод Скрипт()
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
    Урок.ВставитьЕслиОтсутствует("Название", "Физика")
;
```

В результате соответствие **Урок** будет содержать значение **{"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}**.

Получить элемент, если он есть

Часто бывает так, что соответствие создаётся в одном месте, а используете вы его совсем в другом. Заранее неизвестно, будут в нём все ключи, которые вы ожидаете, или нет.

В примере с уроком может быть такая ситуация. Не всегда заранее известна фамилия учителя, который будет вести урок. Может быть, он заболел, его уже удалили из расписания, но другого учителя вместо него пока не назначили.

Ваш скрипт этого не знает. Он ожидает, что в соответствии фамилия учителя будет.

```
метод Скрипт()
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
```

```
    пер Учитель = Урок["Учитель"]
;
```

Вы запускаете скрипт и получаете следующую ошибку.

```
Соответствие не содержит значения по ключу "Учитель"
Scripts::скрипт.Скрипт[3]
```

Для таких случаев есть два метода. Во-первых, метод `Соответствие.ПолучитьИлиНеопределено()`. Этот метод получает значение по указанному ключу, а если такого ключа нет, то возвращает значение `Неопределено`. Вам остаётся только самостоятельно обработать эту ситуацию.

```
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
    пер Учитель = Урок.ПолучитьИлиНеопределено("Учитель")
    если Учитель == Неопределено
        Консоль.Записать("В соответствии нет элемента с ключом \"Учитель\"")
    ;
```

В этом примере, если указанного ключа в соответствии нет, в терминал выводится сообщение.

Во-вторых, есть метод `Соответствие.ПолучитьИлиУмолчание()`. Этот метод также получает значение по указанному ключу, а если такого ключа нет, то возвращает значение, которое указано во втором аргументе.

```
метод Скрипт()
    пер Урок = {"Название":"Математика", "НомерКабинета":311, "Начало":ДатаВремя{2025-09-02 09:00:00}}
    пер Учитель = Урок.ПолучитьИлиУмолчание("Учитель", "Не назначен")
    Консоль.Записать("Учитель - " + Учитель)
;
```

В этом примере, если указанного ключа в соответствии нет, в терминал выводится сообщение **Учитель - Не назначен**. То есть, если из таких соответствий вы формируете расписание занятий, там, где учитель назначен, будет написано **Учитель - Сергеев**, например. Там, где нет преподавателя, будет сразу же написано **Учитель - Не назначен**.

Сравнение соответствий

Соответствия можно сравнивать с помощью операции сравнения на равенство `==`. Два соответствия являются равными, если:

- равны их размеры;
- ключи одного соответствия являются ключами другого;
- попарно совпадают все значения одинаковых ключей.

Порядок следования элементов не имеет значения.

Следующие соответствия равны.

```
{"1-й этап":120, "2-й этап":"3 часа", "3-й этап":310}
{"3-й этап":310, "1-й этап":120, "2-й этап":"3 часа"}
```

Следующие соответствия не равны.

```
{"1-й этап":120, "2-й этап":"3 часа", "3-й этап":310}
{"1-й этап":120, "2-й этап":"3 часа", "3-й этап":310, "4-й этап":4}
```

42. Задание простое

Создайте с помощью литерала соответствие, в котором будет храниться ваш домашний адрес: улица, номер дома, корпус, квартира, город, район, область, край, республика. Создайте в соответствии только те элементы, которые есть в вашем адресе. Решение смотрите здесь *(304)*.

43. Задание простое

В отладке посмотрите значения отдельных элементов вашего соответствия. Например, улица и город. Решение смотрите здесь *(304)*.

44. Задание простое

Используйте такое соответствие.

```
пер Адрес = {"Квартира":36, "Дом":37, "Улица":"Можайское шоссе", "Город":"Одинцово", "Область":"Московская"}
```

В одной переменной сформируйте адрес вашего населённого пункта, а в другой — ваш адрес в этом населённом пункте. Решение смотрите здесь *(305)*.

45. Задание простое

Создайте соответствие, в котором будут содержаться дни рождения двоих ваших самых близких друзей или родственников. Создайте и заполните это соответствие только с помощью конструктора. Для указания дней рождений используйте тип `Дата`. Решение смотрите здесь *(305)*.

46. Задание сложное

Используйте такое соответствие.

```
пер Адрес = {"Квартира":36, "Дом":37, "Улица":"Можайское шоссе", "Город":"Одинцово", "Область":"Московская"}
```

Например, это соответствие может содержать корпус дома, а может и не содержать его. Сформируйте ваш адрес в населённом пункте так, чтобы скрипт работал без ошибок как в одном, так и в другом случае. Решение смотрите здесь *(305)*.

Примечание:

Работа с соответствиями на углублённом уровне описана здесь *(199)*.

Примечание:

- О работе с соответствиями читайте в руководстве разработчика.
- Описание типа `Соответствие` читайте в справке по стандартной библиотеке.

Множество

Общая информация

Множество — это неупорядоченная неименованная коллекция уникальных элементов. В ней нет порядка и нет никакого признака, по которому можно было бы отличить один элемент от другого. Важно понимать, что в множестве не может быть двух одинаковых элементов, поэтому все методы, добавляющие что-то во множество, не добавят в него те элементы, которые там уже есть.

Литерал и конструктор

Литерал множества может выглядеть следующим образом:

```
пер КраткийЛитерал = {1, 2, 3}
```

Внутри фигурных скобок вы перечисляете значения элементов множества. В данном случае это три числа **1**, **2** и **3**. Поскольку из литерала понятны типы элементов множества, описание типа можно не писать.

Однако может случиться так, что в литерале вы написали числа, но в будущем планируете добавлять в это множество и строки тоже. В этом случае нужно перечислить перед фигурными скобками «{» «}» внутри угловых скобок «<» «>» все типы, которые могут принимать элементы множества. Типы перечисляются через вертикальную черту «|».

Например, литерал множества, которое может содержать числа и строки.

```
пер ЛитералСТипом = <Число|Строка>{1, 2, 3}
```

> **Примечание:**
>
> Фигурные «{» «}» и угловые скобки «<» «>», а также вертикальную черту «|» можно вводить с помощью Alt-ввода (227).

С помощью литерала вы можете описать пустое множество. В этом случае указание типа его элементов является обязательным. Например, пустое множество, которое может содержать строки.

```
пер ЛитералПустогоМножества = <Строка>{}
```

Множество имеет два конструктора. Один конструктор создаёт пустое множество, а другой создаёт новое множество копированием существующего.

Пустое множество чисел будет выглядеть следующим образом.

```
пер ПустоеМножествоЧисло = новый Множество<Число>()
```

Добавить элемент

Чтобы добавить элемент в множество, используйте метод `Множество.Добавить()`.

```
пер Числа = {1, 2, 3, 5, 8}
Числа.Добавить(13)
```

В результате множество **Числа** будет содержать значение **{1, 2, 3, 5, 8, 13}**.

Доступ к элементам множества

Доступ к элементам множества поддерживается только с помощью цикла `для из`. Несмотря на то что множество — неупорядоченная коллекция, элементы будут обходиться в порядке их добавления в множество.

```
метод Скрипт()
    пер Числа = {1, 2, 3, 5, 8}
    для Число из Числа
        Консоль.Записать(Число)
    ;
;
```

В результате в терминал будет выведен следующий текст:

```
1
2
3
5
8
```

Удалить элемент

Чтобы удалить элемент множества, используйте метод `Множество.Удалить()`:

```
пер Числа = {1, 3, 5, 7, 9}
Числа.Удалить(9)
```

В результате множество **Числа** будет содержать значение **{1, 3, 5, 7}**.

Чтобы удалить сразу все элементы множества, используйте метод `Множество.Очистить()`.

Добавить все элементы другого множества

Это может понадобиться для формирования списка адресатов. Некоторые адресаты уже были добавлены в множество вручную. Теперь надо туда же добавить всех сотрудников конкретного отдела, которые содержатся в другом множестве.

Для этого используйте метод `Множество.ДобавитьВсе()`:

```
пер СписокАдресатов = {"Алексеев В.П.", "Лужина Е.А.", "Морозов С.М."}
пер ОтделЗакупок = {"Александрова С.С.", "Ильин С.А.", "Королева Н.П.", "Лужина Е.А."}
СписокАдресатов.ДобавитьВсе(ОтделЗакупок)
```

В результате множество **СписокАдресатов** будет содержать значение **{Алексеев В.П., Лужина Е.А., Морозов С.М., Александрова С.С., Ильин С.А., Королева Н.П.}**.

Здесь в методе `Множество.ДобавитьВсе()` указан единственный аргумент — коллекция, значения которой надо добавить в исходное множество. Это может быть как множество, так и массив, но при этом тип элементов добавляемой коллекции должен совпадать с типом элементов исходного множества.

Получить объединение двух множеств

Это может понадобиться для формирования списка рассылки.

Адресаты добавляются отделами или рабочими группами. При этом в результирующем списке должны содержаться уникальные адресаты, без дублирования.

Для этого используйте метод `Множество.Объединение()`:

```
пер ОтделПродаж = {"Алексеев В.П.", "Морозов С.М.", "Яковлева И.Н.", "Ильин С.А."}
пер Бухгалтерия = {"Александрова С.С.", "Ильин С.А.", "Королева Н.П.", "Яковлева И.Н."}
пер СписокРассылки = ОтделПродаж.Объединение(Бухгалтерия)
```

В результате множество **СписокРассылки** будет содержать значение **{Алексеев В.П., Морозов С.М., Яковлева И.Н., Ильин С.А., Александрова С.С., Королева Н.П.}**.

Здесь в методе `Множество.Объединение()` указан единственный аргумент — множество, с которым нужно объединить исходное множество (для которого вызван метод).

Получить пересечение двух множеств

Это может понадобиться для получения списка совместителей — людей, которые числятся сразу в нескольких отделах или рабочих группах.

Для этого используйте метод `Множество.Пересеченение()`:

```
пер ОтделПродаж = {"Алексеев В.П.", "Морозов С.М.", "Яковлева И.Н.", "Ильин С.А."}
пер Склад = {"Александрова С.С.", "Ильин С.А.", "Королева Н.П.", "Яковлева И.Н."}
пер Совместители = ОтделПродаж.Пересечение(Склад)
```

В результате множество **Совместители** будет содержать значение **{Яковлева И.Н., Ильин С.А.}**.

Здесь в методе `Множество.Пересеченение()` указан единственный аргумент — множество, с которым нужно получить пересечение исходного множества (для которого вызван метод).

Получить разность двух множеств

Это может понадобиться для подбора адресатов письма. Например, нужно показать для подбора всех адресатов кроме тех, которые уже выбраны в письме.

Для этого используйте метод `Множество.Разность()`:

```
пер ВсеСотрудники = {"Алексеев В.П.", "Морозов С.М.", "Яковлева И.Н.", "Ильин С.А."}
пер ВыбранныеАдресаты = {"Александрова С.С.", "Ильин С.А.", "Королева Н.П.", "Яковлева И.Н."}
пер Адресаты = ВсеСотрудники.Разность(ВыбранныеАдресаты)
```

В результате множество **Адресаты** будет содержать значение **{Алексеев В.П., Морозов С.М.}**.

Здесь в методе `Множество.Разность()` указан единственный аргумент — множество, с которым нужно получить разность исходного множества (для которого вызван метод).

Примечание:

- О работе с множествами читайте в руководстве разработчика.
- Описание типа `Множество` читайте в справке по стандартной библиотеке.

Структура

Общая информация

Структура — это ваш собственный тип данных, который вы можете создать в своём скрипте, дать ему имя. В дальнейшем вы можете использовать его как любой другой тип: создавать экземпляры, присваивать их переменным, передавать в качестве аргументов в другие методы и возвращать обратно.

Что представляет собой структура? Это набор полей, в которые вы можете поместить нужные вам значения. Эти значения будут доступны в экземпляре структуры как его свойства. Какие это будут поля, какие у них будут типы значений — вы описываете самостоятельно.

Например, структура может иметь имя **Товар** и описывать один товар следующими полями: **Наименование**, **Артикул** и **Цена**.

Важным моментом является то, что, состав полей структуры и их типы определяются на этапе написания скрипта. Во время исполнения скрипта вы не можете изменить ни состав полей, ни их типы.

Объявление структуры

Структура объявляется в начале скрипта, до методов. Объявление начинается с ключевого слова `структура` и заканчивается точкой с запятой «;». После ключевого слова идёт имя структуры и описываются её поля.

Поля описываются аналогично тому, как объявляются переменные *(23)*: модификатор, имя и тип. Дополнительно можно указать значение инициализации и новое ключевое слово `обз`. Например:

```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
    пер Цена: Число = 1
;
```

В этом примере объявляется структура с именем **Товар**, у которой есть три поля: **Наименование**, **Артикул** и **Цена**.

Наименование товара изменять нельзя (модификатор `знч`), а цена товара, если не указана при создании, будет иметь значение 1 по умолчанию.

Поле **Наименование** имеет модификатор обязательности `обз`. Это значит, что оно обязательно должно быть обязательно указано, когда вы будете конструировать экземпляр этой структуры.

Конструктор

«1С:Элемент» автоматически создаёт для вашей структуры конструктор, который включает в себя все поля структуры. Значения полей, которые вы отметили как `обз`, обязательно должны быть указаны в конструкторе, значения остальных полей — по желанию.

Например, вы можете создать два экземпляра своей структуры **Товар**:

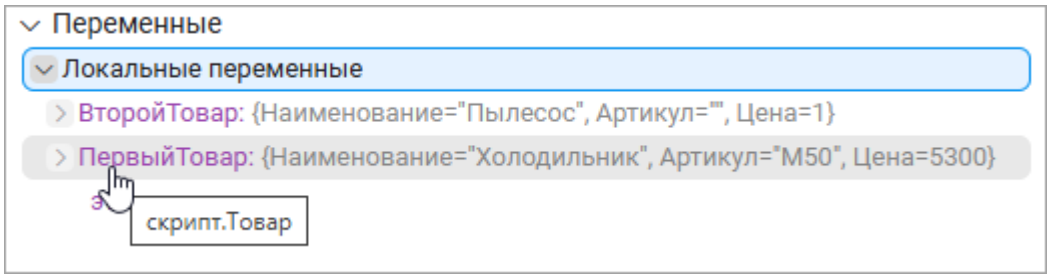
```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
    пер Цена: Число = 1
;

метод Скрипт()
    пер ПервыйТовар = новый Товар("Холодильник", "M50", 5300)
    пер ВторойТовар = новый Товар("Пылесос")
;
```

При инициализации переменной **ПервыйТовар** использована полная форма конструктора, в которой указаны значения всех полей, при инициализации переменной **ВторойТовар** — краткая форма, в которой указано значение только обязательного поля **Наименование**.

> **Примечание:**
>
> Вы можете в обязательном порядке использовать в конструкторе только именованные аргументы. Как это сделать, смотрите здесь *(210)*.

Просмотр структуры

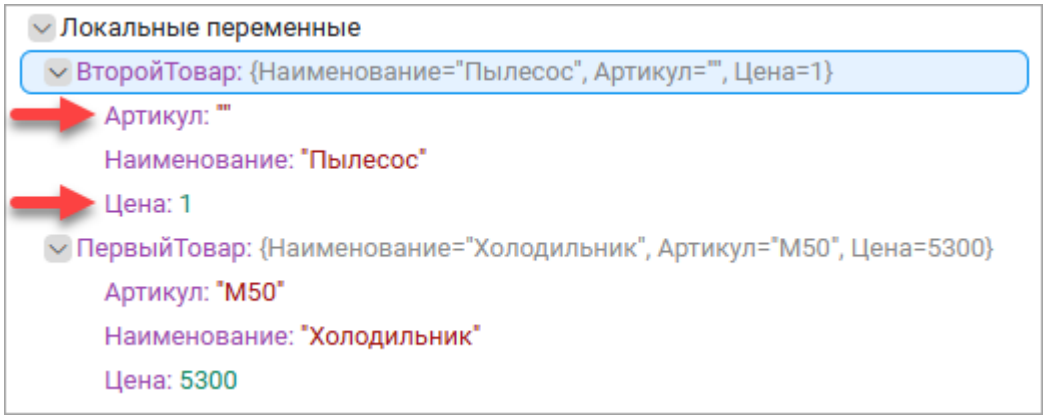
Установите точку останова на последней строке скрипта и запустите отладку.



Вы видите, что у вас есть две переменные, которые имеют интересный тип `скрипт.Товар`. Именно так обозначается тип созданной вами структуры. Имя файла вашего скрипта **скрипт** говорит о том, что это не тип стандартной библиотеки «1С:Элемента», как было до этого: `Стд::Строка`, `Стд:Число`, `Стд:ДатаВремя` или `Стд::Коллекции::Массив<Число>`. Это некий тип, который определён только в области видимости вашего скрипта.

После точки следует имя вашей структуры — **Товар**. Из этого понятно, что имя структуры должно быть уникально в пределах вашего скрипта.

Разверните эти переменные.

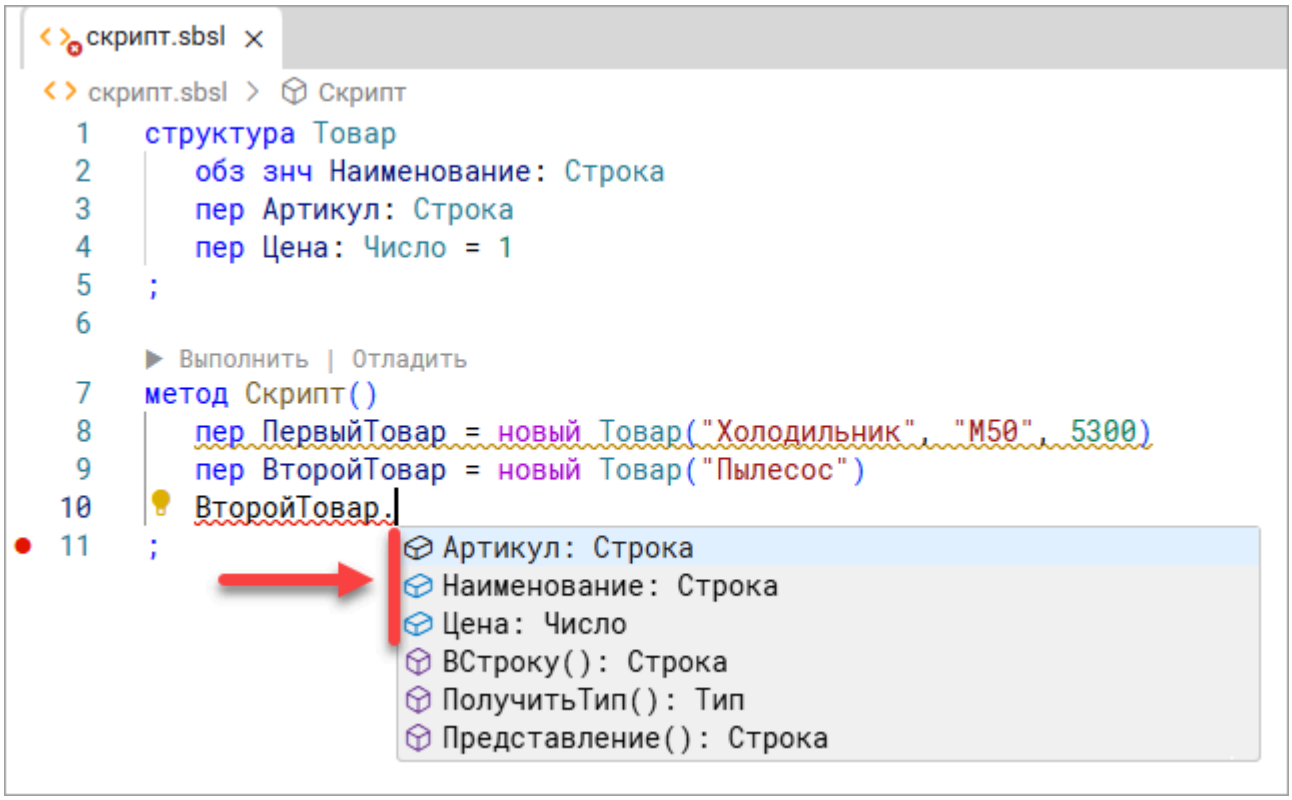


Вы видите, что у первого товара заполнены все поля, а у второго поле **Цена** имеет значение инициализатора, **1**, а поле **Артикул** имеет значение по умолчанию типа `Строка` — пустая строка. Значения этих полей вы не указывали при конструировании второго товара.

Контекстная подсказка и обращение к полям экземпляра

Завершите отладку и вернитесь в скрипт. Сейчас вы добавите артикул и цену второму товару.

Напишите **ВторойТовар** и поставьте точку.



Контекстная подсказка подскажет вам, что вы можете использовать три свойства: **Артикул**, **Наименование** и **Цена**. Это те поля, которые вы описали для своей структуры. Они доступны как свойства экземпляра.

Используя эти свойства, вы можете установить значения артикула и цены второго товара.

```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
```

```
    пер Цена: Число = 1
;

метод Скрипт()
    пер ПервыйТовар = новый Товар("Холодильник", "M50", 5300)
    пер ВторойТовар = новый Товар("Пылесос")
    ВторойТовар.Артикул = "VC20"
    ВторойТовар.Цена = 5000
;
```

Посмотрите в отладке, как изменилось значение второго товара.

Методы структуры

Помимо полей, которые доступны как свойства экземпляра, вы можете описать и собственные методы структуры. Эти методы будут доступны у экземпляров структуры.

Для какой цели могут понадобиться такие методы? Например, для того, чтобы увеличить цену товара на указанное количество процентов.

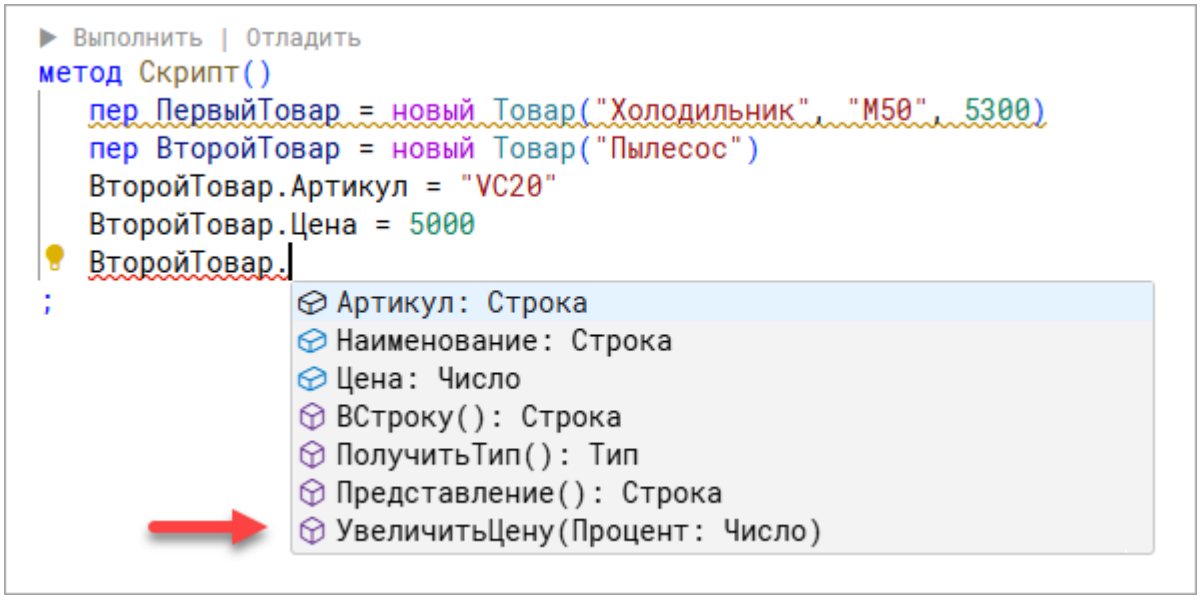
Методы структуры описываются в объявлении структуры. Описание метода структуры выглядит точно так же, как и описание «обычного» метода, который вы изучали раньше *(81)*.

```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
    пер Цена: Число = 1

    метод УвеличитьЦену(Процент: Число)
        Цена += Цена/100*Процент
    ;
;

метод Скрипт()
    пер ПервыйТовар = новый Товар("Холодильник", "M50", 5300)
    пер ВторойТовар = новый Товар("Пылесос")
    ВторойТовар.Артикул = "VC20"
    ВторойТовар.Цена = 5000
;
```

В этом примере добавлен метод **УвеличитьЦену**, который получает число процентов, на которое нужно увеличить цену, и увеличивает её.

Поля структуры доступны в её методах по своим именам, в данном случае используется значение поля **Цена**.

Если теперь вы напишете в модуле **ВторойТовар.** — то увидите, что контекстная подсказка сразу же предлагает использовать вам метод **УвеличитьЦену()**.



Увеличьте цену товара на 20 %. Посмотрите в отладке, что цена второго товара вместо 5 000 стала 6 000.

Статические методы

Помимо простых методов структура может иметь статические методы. Простые методы, как вы знаете, доступны у экземпляров типа. Чтобы воспользоваться простым методом, вы должны сначала создать экземпляр типа — с помощью литерала или с помощью конструктора. После этого вы можете обратиться к этому экземпляру и выполнить простой метод. Очевидно, что простой метод использует в своей работе какие-то данные этого конкретного экземпляра.

Статический метод — это метод, принадлежащий типу, а не экземпляру. Для выполнения статического метода не нужен конкретный экземпляр, не нужны данные конкретного экземпляра. Достаточно указать имя типа и через точку от него имя статического метода. Это не будет настоящее обращение через точку, как к свойствам или методам экземпляра, потому что нет никакого экземпляра. Это просто такая форма записи обращения к статическому методу.

«Что может делать статический метод, если ему не нужны данные экземпляра?» — спросите вы. Например, в случае со структурой он может стать неким «особенным» конструктором, который «заточен» под конкретную практическую задачу.

Например, структура **Товар** используется для обработки списков товаров, поступающих к вам в виде текстовых файлов. В таких файлах каждая строка содержит значения наименования, артикула и цены, разделённые точкой с запятой «;». Вот пример такого файла **items.txt**.

```
Холодильник;M50;5300
Чайник;Z23-1;2800
Стиральная  машина;8E30;35000
```

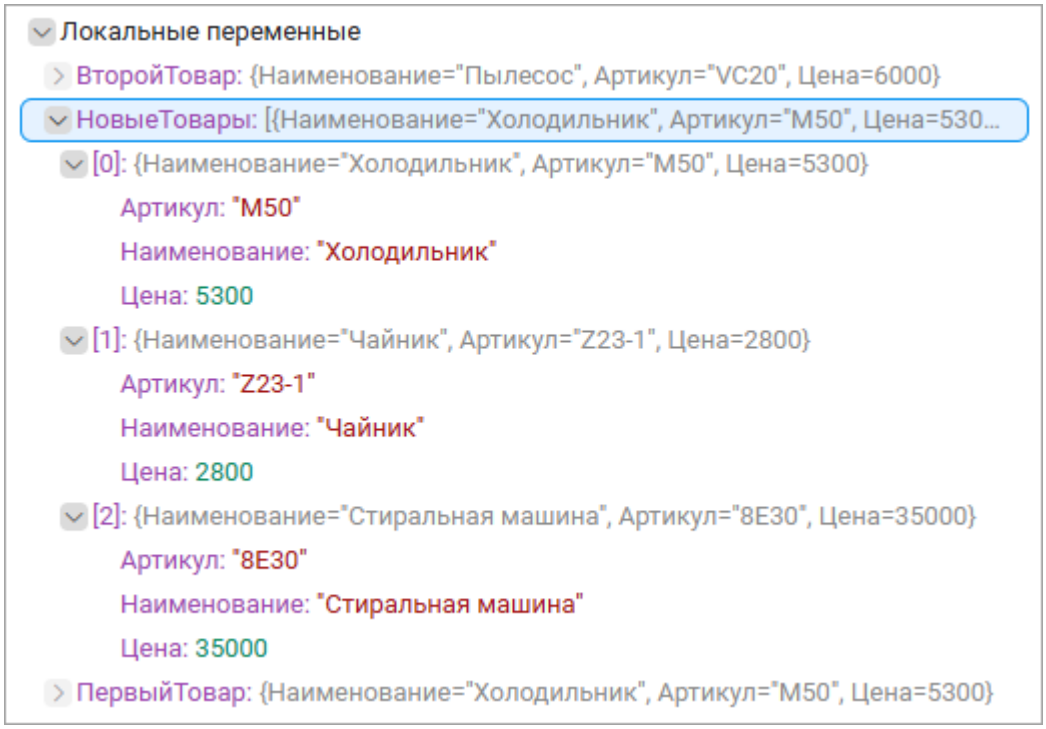
Нужно прочитать этот файл и создать массив, который будет содержать описания этих товаров в виде экземпляров вашей структуры **Товар**.

Для этой задачи хорошо подходит статический метод, который требует только указания типа вашей структуры. Статические методы структуры описываются так же, как и простые, только в начале добавляется ключевое слово `статический`.

Пока вы не знакомились с чтением и обработкой *(250)* текстовых файлов, поэтому просто скопируйте себе текст скрипта, запишите на диск **D:** файл **items.txt**.

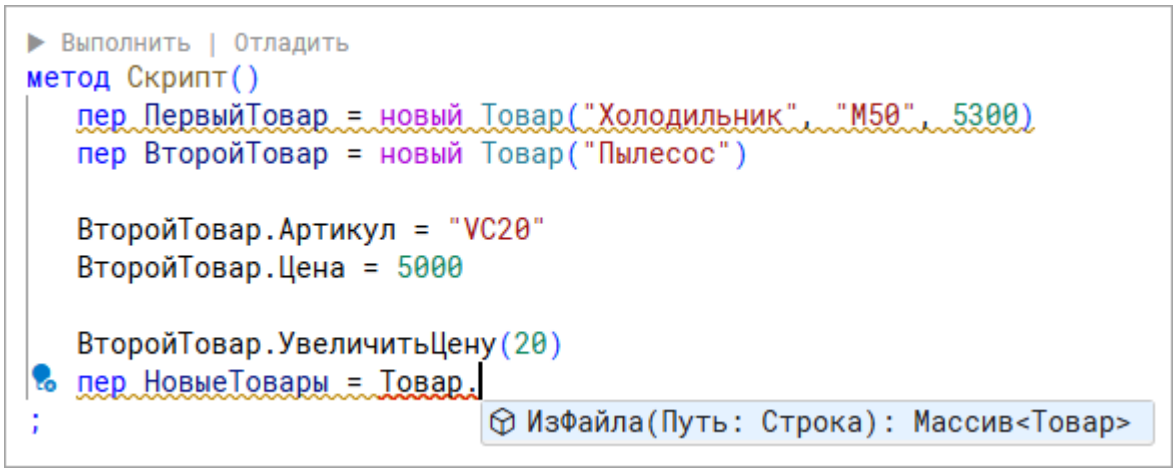
```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
    пер Цена: Число = 1

    метод УвеличитьЦену(Процент: Число)
        Цена += Цена/100*Процент
    ;

    статический метод ИзФайла(Путь: Строка): Массив<Товар>
        пер Строки: Массив<Товар>
        исп ПотокЧтения = новый Файл(Путь).ОткрытьПотокЧтения()
        знч ЧтениеДанных = новый ЧтениеДанных(ПотокЧтения)
        пока не ЧтениеДанных.ЧтениеЗавершено()
            знч Строка = ЧтениеДанных.ПрочитатьСтроку()
            знч Части = Строка.Разделить(";")
            Строки.Добавить(новый Товар(Части[0], Части[1], новый Число(Части[2])))
        ;
        возврат Строки
    ;
;

метод Скрипт()
    пер ПервыйТовар = новый Товар("Холодильник", "M50", 5300)
    пер ВторойТовар = новый Товар("Пылесос")

    ВторойТовар.Артикул = "VC20"
    ВторойТовар.Цена = 5000

    ВторойТовар.УвеличитьЦену(20)
    пер НовыеТовары = Товар.ИзФайла("D:\\items.txt")
;
```

Установите точку останова в последней строке, запустите отладку и посмотрите, как это работает.



В переменной **НовыеТовары** появится массив с товарами из файла.

Попробуйте написать вызов статического метода самостоятельно. Вы увидите, что контекстная подсказка, во-первых, предлагает написать имя вашей структуры, а потом, через точку, предлагает написать имя статического метода.



Как читать и записывать *(250)* файлы вы узнаете позже, но вот вам сейчас короткое объяснение того, что написано в статическом методе.

Переменная **Строки** — это тот массив товаров, который нужно вернуть. Переменные **Файл**, **ПотокЧтения** и **ЧтениеДанных** — это специальные типы для чтения и записи файлов.

Сначала по переданному пути **Путь** создаётся файл, из него открывается поток чтения. Из потока чтения создаётся чтение данных, которое читает по одной строке файла в цикле.

```
исп ПотокЧтения = новый Файл(Путь).ОткрытьПотокЧтения()
знч ЧтениеДанных = новый ЧтениеДанных(ПотокЧтения)
пока не ЧтениеДанных.ЧтениеЗавершено()
```

```
знч Строка = ЧтениеДанных.ПрочитатьСтроку()
....
```

Каждая прочитанная строка делится на части:

```
знч Части = Строка.Разделить(";")
```

Части — это массив. Из его элементов конструктором создаётся новый экземпляр структуры **Товар**:

```
новый Товар(Части[0], Части[1], новый Число(Части[2]))
```

Этот экземпляр добавляется в возвращаемый массив **Строки**:

```
Строки.Добавить(...)
```

Иерархия типов

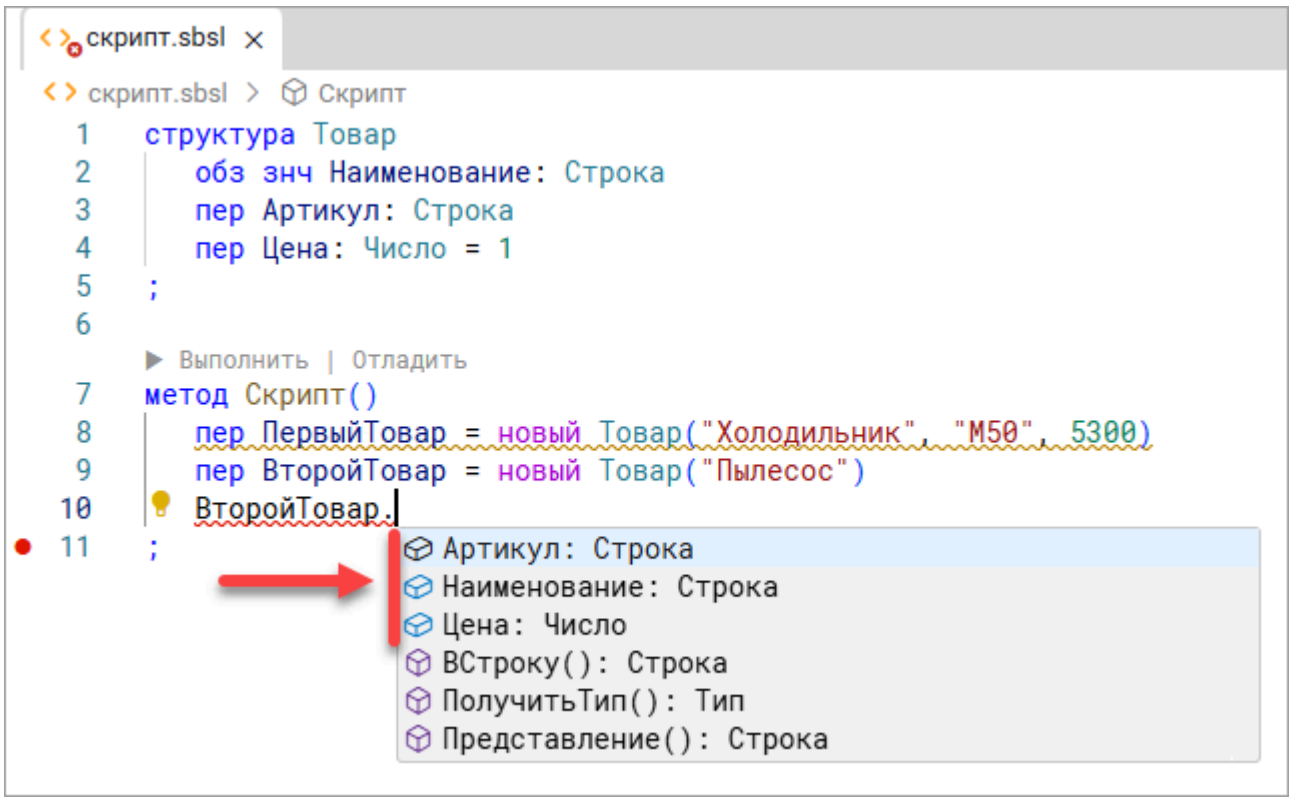
Тип структуры, несмотря на то что вы создаёте его сами в своём скрипте, не существует в «безвоздушном» пространстве, он встроен в общую систему типов «1С:Элемента».

Если вам интересно, вы можете скопировать в свой скрипт фрагмент кода *(104)*, который изучали, когда знакомились с иерархией типов, и посмотреть базовые типы вашей структуры.

Базовым типом для неё является `Конструируемое`, а для него, в свою очередь, `Объект`.

```
скрипт.Товар -> Стд::Конструируемое -> Стд::Объект
```

Именно поэтому, когда вы вызывали контекстную подсказку для экземпляра структуры, она показывала вам три метода, унаследованные от типа `Объект`: **ВСтроку()**, **ПолучитьТип()** и **Представление()**.



> **Примечание:**
>
> О работе со структурами читайте в руководстве разработчика.

Перечисление

Общая информация

Перечисление — это ваш собственный тип данных, который вы можете создать в своём скрипте, дать ему имя. Перечисление представляет собой явно заданный набор идентификаторов. Эти идентификаторы можно использовать для помещения в переменные или поля других экземпляров.

Например, перечисление может иметь имя **ВидСообщения** и содержать следующие идентификаторы: **Важное**, **Обычное** и **Второстепенное**. Эти идентификаторы можно использовать при создании сообщений, указывая важность сообщений.

Объявление перечисления

Перечисление объявляется в начале скрипта, до методов. Объявление начинается с ключевого слова `перечисление` и заканчивается точкой с запятой «;». После ключевого слова идёт имя перечисления и описываются его значения-идентификаторы.

Например:

```
перечисление  ВидСообщения
    Важное,
    Обычное  умолчание,
    Второстепенное
;
```

В этом примере описано перечисление с именем **ВидСообщения** с тремя элементами: **Важное**, **Обычное** и **Второстепенное**. Элемент **Обычное** отмечен как элемент по умолчанию.

В именах перечислений рекомендуется использовать слово **Вид** и не использовать слово **Тип**.

В перечислении должно быть хотя бы одно значение.

Перечисление может иметь одно значение по умолчанию или вовсе не иметь значения по умолчанию. Значение по умолчанию нужно «1С:Элементу» тогда, когда переменная описывается с типом создаваемого перечисления и для этой переменной не указывается значение инициализации.

Использование

Скопируйте себе следующий пример.

```
перечисление  ВидСообщения
    Важное,
    Обычное  умолчание,
    Второстепенное
;
```

```
структура Сообщение
    пер Адресат: Строка
    пер Текст: Строка
    пер Важность: ВидСообщения
;

метод Скрипт()
    пер НовоеСообщение = новый Сообщение("Получатель", "Проверка работы", ВидСообщения.Важное)
;
```

В этом примере объявляется перечисление **ВидСообщения** и объявляется структура **Сообщение**. Экземпляр структуры содержит одно сообщение, у которого есть адресат, текст и важность. Так вот, важность сообщения задаётся значением перечисления **ВидСообщения**.

В описании типа поля **Важность** просто указывается имя типа перечисления:

```
пер Важность: ВидСообщения
```

При формировании сообщения в конструкторе структуры значение важности указывается как имя перечисления и, через точку, нужное значение:

```
ВидСообщения.Важное
```

Методы перечисления

У перечисления, как и у структуры *(143)*, могут быть методы. Синтаксис описания методов тот же самый.

Единственный вопрос, который может у вас возникнуть: «Зачем вообще перечислению методы?»

Например, перечисление называется **РекомендуемыеЦвета**. Оно содержит фиксированный набор цветов, которые нужно использовать для оформления отчётов. В этом перечислении цвета обозначаются понятным для человека образом: **Красный**, **Желтый**, **Синий**.

```
перечисление РекомендуемыеЦвета
    Красный,
    Желтый,
    Синий

    метод RGB(): АбсолютныйЦвет
        пер АбсолютныйЦвет = новый АбсолютныйЦвет(0, 0, 0)
        выбор этот
        когда Красный
            АбсолютныйЦвет = новый АбсолютныйЦвет(244, 124, 113)
        когда Желтый
            АбсолютныйЦвет = новый АбсолютныйЦвет(252, 246, 136)
        когда Синий
            АбсолютныйЦвет = новый АбсолютныйЦвет(96, 141, 200)
        ;
        возврат АбсолютныйЦвет
    ;
;

метод Скрипт()
    пер МойКрасный = РекомендуемыеЦвета.Красный
```

```
    пер RGB = МойКрасный.RGB()
;
```

Это хорошо до тех пор, пока не наступает необходимость нарисовать что-то на экране в соответствии с этими цветами. Тут оказывается, что красный цвет имеет множество оттенков и неизвестно, какой из них нужно использовать. Так же и с остальными цветами.

Вот на этот случай у перечисления **РекомендуемыеЦвета** описан метод **RGB()**. Он возвращает точный абсолютный цвет для каждого из значений этого перечисления.

В методе используется ключевое слово `этот`. Оно обозначает тот элемент перечисления, для которого вызывается метод.

Статические методы

У перечисления, как и у структуры *(144)*, могут быть статические методы. Синтаксис описания методов тот же самый.

Статический метод в перечислении может понадобиться для выделения части значений перечисления. Например, ваше перечисление содержит наименования операционных систем. Эти значения используются в заявках, которые поступают от пользователей. В зависимости от того, при работе в какой операционной системе обнаружена проблема, заявку нужно направить той или иной команде разработчиков.

Неудобство заключается в том, что «линуксов», например, существует большое количество, но всеми ими занимается команда Linux-разработчиков.

В этой ситуации вам поможет статический метод, который позволит получить только те значения перечисления, которые относятся к семейству Linux. Если ваше значение входит в их число, значит, заявку нужно отправить Linux-разработчикам.

Скопируйте себе следующий пример.

```
перечисление ОС
    Android,
    iOS,
    macOS,
    WindowsServer,
    Windows умолчание,
    Debian,
    Ubuntu,
    Альт

    статический метод Windows(): Массив<ОС>
        возврат [ОС.Windows, ОС.WindowsServer]
    ;

    статический метод Linux(): Массив<ОС>
        возврат [ОС.Debian, ОС.Ubuntu, ОС.Альт]
    ;

    статический метод Мобильная(): Массив<ОС>
        возврат [ОС.Android, ОС.iOS]
    ;
```

```
;

метод Скрипт()
    пер НоваяОС = ОС.Debian
    если ОС.Windows().Содержит(НоваяОС)
        // Отправить разработчикам Windows
        Консоль.Записать("Windows")
    иначе если ОС.Linux().Содержит(НоваяОС)
        // Отправить разработчикам Linux
        Консоль.Записать("Linux")
    ;
;
```

В результате в терминал будет выведено сообщение **Linux**.

В этом примере объявлено перечисление **ОС**, содержащие названия различных операционных систем.

У перечисления есть три статических метода, которые возвращают массив значений этого перечисления. В одном случае это операционные системы, относящиеся к семейству Windows, в другом — к Linux, а в третьем — относящиеся к мобильным операционным системам.

В переменной **НоваяОС** находится значение этого перечисления, полученное из очередной заявки. В инструкции `если` проверяется, содержится ли это значение в массиве ОС Windows или ОС Linux:

```
если ОС.Windows().Содержит(НоваяОС)
```

В зависимости от этого заявка отправляется той или иной команде разработчиков.

Иерархия типов

Тип перечисления, несмотря на то что вы создаёте его сами в своём скрипте, не существует в «безвоздушном» пространстве, он встроен в общую систему типов «1С:Элемента».

Цепочка базовых типов для него выглядит так:

```
скрипт.ОС -> Стд::Перечисление -> Стд::Представляемое -> Стд::Объект
```

Глава 6. Углублённый уровень

Составные типы

Общая информация

«1С:Элемент» использует статическую типизацию, и для каждой переменной, параметра и возвращаемого значения заранее известны их типы. Однако порой возникают задачи, когда в одной переменной нужно хранить значения разных типов. Например, цена товара в прайс-листе может быть указана числом **543.23** или строкой **"Договорная"**.

В таких случаях вы можете описать *составной тип*, в котором перечислить все типы, которые может принимать переменная.

Вы сможете помещать в эту переменную значения перечисленных типов, и в каждый конкретный момент тип этой переменной будет соответствовать тому значению, которое в ней находится.

Описание составного типа

В описании составного типа все имена типов указываются друг за другом через вертикальную черту «|» без пробелов. Например:

```
пер МояПеременная: Строка|Число|Булево
```

В этом примере указано, что **МояПеременная** может принимать значения типов `Строка`, `Число` и `Булево`.

> **Примечание:**
>
> Вертикальную черту «|» можно вводить с помощью Alt-ввода *(227)*.

Инициализатор переменной составного типа

Как вы знаете, из инструкции объявления переменной должно быть понятно её начальное значение *(26)*.

В случае составного типа тут возникает проблема. Даже если типы, входящие в составной, имеют собственные значения по умолчанию, непонятно, какое из этих значений и почему должно стать начальным значением для переменной.

Поэтому для переменной составного типа нужно в явном виде указывать инициализатор.

```
пер МояПеременная: Строка|Число|Булево = 7
```

В результате в переменной **МояПеременная** будет значение **7** и она будет иметь тип `Число`.

В то же время есть способ не указывать инициализатор для переменной составного типа. Об этом рассказывается в следующем разделе.

Тип «Неопределено» и значение «Неопределено»

Инициализатор *(152)* переменной составного типа удобен в тех случаях, когда вы хотите сразу присвоить переменной одно из «рабочих» значений. Но что, если вам нужно только лишь создать переменную, а «работать» с ней вы начнёте позже и где-то в другом месте?

Что, если вам важно отличать состояние этой переменной: произошёл уже в результате работы вашего алгоритма какой-то выбор её типа или она ещё не использовалась? Может быть, значение этой переменной должен вводить пользователь, и тогда перед первым использованием ему нужно сообщить, например, о том, что в эту переменную можно ввести как строку, так и число?

Специально для этих целей в «1С:Элементе» существует тип `Неопределено`, который имеет единственное значение, обозначаемое литералом **Неопределено**.

Если в составе составного типа есть тип `Неопределено`, то переменная принимает значение **Неопределено**. В описании типа тип `Неопределено` обозначается вопросительным знаком «?» для краткости.

```
метод Скрипт()
    пер МояПеременная: Строка|Число|?
;
```

В результате в переменной **МояПеременная** будет значение **Неопределено** и она будет иметь тип `Неопределено`.

Если в составном типе кроме `Неопределено` присутствует только один тип, тогда для краткости в описании типа перед вопросительным знаком «?», обозначающим `Неопределено`, можно не писать вертикальную черту «|».

```
метод Скрипт()
    пер МояПеременная: Строка?
;
```

Операция «как»

Порой тип значения переменной составного типа нужно привести к одному или нескольким возможным типам.

Например, есть **МояПеременная**, которая имеет составной тип `Строка|Число`. В ней находится строковое значение **"тест"** и вы хотите узнать длину этой строки. Для этого нужно выполнить метод `Строка.Длина()`, но проблема заключается в том, что у типа `Число`, которым также может быть переменная **МояПеременная**, такого метода нет.

По этой причине компилятор «1С:Элемента» сообщит вам об ошибке: **Неизвестный метод "Представляемое.Длина"**.

```
метод Скрипт()
    пер МояПеременная: Строка|Число = "тест"
    пер ДлинаСтроки = МояПеременная.Длина()
;
```

Чтобы этого не происходило, нужно тип значения переменной **МояПеременная** привести к типу `Строка`. Для этого можно использовать операцию `как`:

```
метод Скрипт()
    пер МояПеременная: Строка|Число = "тест"
    пер ДлинаСтроки = (МояПеременная как Строка).Длина()
;
```

В результате в переменной **ДлинаСтрока** будет значение **4**.

Результирующий тип должен входить в состав составного типа

Операция `как` не может привести тип значения к любому типу, который вам захочется. Результирующий тип должен быть одним из тех, которые перечислены в составе исходного составного типа.

Например, в следующих примерах тип значения будет успешно приведён, т.к. типы `Строка` и `Число` присутствуют в исходном составном типе.

```
метод Скрипт()
    пер МояПеременная: Строка|Число|Булево = 123
    пер Переменная1 = (МояПеременная как Строка|Число)
    пер Переменная2 = (МояПеременная как Число)
;
```

В то же время в следующем примере будет получена ошибка компиляции **Несовместимые типы "Булево|Число|Строка" и "ДатаВремя"**, потому что тип **ДатаВремя** не содержится в исходном составном типе.

```
метод Скрипт()
    пер МояПеременная: Строка|Число|Булево = 123
    пер Переменная1 = (МояПеременная как ДатаВремя)
;
```

Нельзя преобразовать значение одного типа в значение другого типа

Значение, которое находится в исходной переменной, нельзя преобразовать в значение другого типа с помощью операции `как`.

Например, следующий код не вызовет ошибок компиляции.

```
метод Скрипт()
    пер МояПеременная: Строка|Число|Булево = 123
    пер Переменная1 = (МояПеременная как Строка|ДатаВремя)
;
```

Однако во время выполнения возникнет следующая ошибка:

```
Значение типа "Number", не соответствует ожидаемому типу "String|DateTime".
```

Дело в том, что операция `как` не может преобразовать значение типа `Число` в значение типа `Строка` или `ДатаВремя`.

В то же время, если в этой же ситуации в переменной **МояПеременная** окажется строковое значение **"раз два три"**, всё выполнится без ошибок.

```
метод Скрипт()
    пер МояПеременная: Строка|Число|Булево = "раз два три"
    пер Переменная1 = (МояПеременная как Строка|ДатаВремя)
;
```

В данном случае тип, который имеет значение, `Строка`, есть и в исходном описании типа, и в результирующем.

Настойчивая операция

Если вы уверены, что выражение в момент выполнения скрипта не может иметь значение **Неопределено**, то после этого выражения можно поставить восклицательный знак «!».

Смысл этой операции заключается только в том, чтобы убрать из описания составного типа тип `Неопределено`, если известно, что его там точно быть не может.

Проиллюстрировать это можно на примере структур (подробнее о структурах рассказывается в разделе Структура *(140)*).

Например, есть две структуры: **Сотрудник** и **ПочтовыйАдрес**.

```
структура Сотрудник
    пер ФИО: Строка
    пер Адрес: ПочтовыйАдрес?
    пер Телефон: Строка?
;

структура ПочтовыйАдрес
    пер Страна: Строка?
    пер Город: Строка
    пер Улица: Строка?
    пер Дом: Строка?
;
```

Структура **Сотрудник** имеет поля **ФИО**, **Адрес** и **Телефон**. В поле **Адрес** содержится, в свою очередь, экземпляр другой структуры — **ПочтовыйАдрес**. Она имеет поля **Страна**, **Город**, **Улица** и **Дом**.

Например, в программном коде создаётся экземпляр сотрудника, и точно известно, что у него есть адрес.

Если просто присвоить этот адрес переменной **АдресСотрудника**, то она будет иметь составной тип `ПочтовыйАдрес?`.

```
метод Скрипт()
    пер Стажер = новый Сотрудник("Иванов", новый ПочтовыйАдрес("Владимир"))
    пер АдресСотрудника = Стажер.Адрес
;
```

Однако, если точно известно, что в переменной **АдресСотрудника** не может быть значения `Неопределено`, можно использовать настойчивую операцию «!», чтобы убрать тип `Неопределено` из описания типов.

```
метод Скрипт()
    пер Стажер = новый Сотрудник("Иванов", новый ПочтовыйАдрес("Владимир"))
    пер АдресСотрудника = Стажер.Адрес!
;
```

В результате переменная **АдресСотрудника** будет иметь тип `ПочтовыйАдрес`.

Операция «Безопасный доступ»

Эту операцию можно проиллюстрировать на примере тех же структур **Сотрудник** и **ПочтовыйАдрес**, которые были рассмотрены выше.

Например, заранее неизвестно, заполнен ли адрес у сотрудника. Если в такой ситуации не предпринять никаких действий, то следующий код приведёт к ошибке во время работы приложения:

```
структура Сотрудник
    пер ФИО: Строка
    пер Адрес: ПочтовыйАдрес?
    пер Телефон: Строка?
;

структура ПочтовыйАдрес
    пер Страна: Строка?
    пер Город: Строка
    пер Улица: Строка?
    пер Дом: Строка?
;

метод Скрипт()
    пер Менеджер = новый Сотрудник("Иванов")
    пер СтранаСотрудника = Менеджер.Адрес.Страна
;
```

В результате вы получите следующее сообщение:

```
Свойство "Страна" не найдено у значения типа "Std::Undefined".
```

В переменной **Адрес** находится значение `Неопределено`, и «1С:Элемент» не может найти у него свойство **Страна**.

Чтобы этого не случилось, можно использовать операцию безопасного доступа «?», с помощью которой выполняется доступ через точку (или вызов метода), только если источник вызова не `Неопределено`, иначе результат всей цепочки становится `Неопределено`.

Для осуществления безопасного доступа после источника вызова поставьте вопросительный знак «?».

```
пер Менеджер = новый Сотрудник("Иванов")
пер СтранаСотрудника = Менеджер.Адрес?.Страна
```

Тогда, если адрес не указан, ошибки не возникнет, а всё выражение (переменная **СтранаСотрудника**) будет иметь значение `Неопределено`, которое можно обработать в алгоритме. Например, оповестить ответственного о том, что необходимо заполнить адрес сотрудника.

Операция «Умолчание»

Эта операция обозначается двумя вопросительными знаками «??» и возвращает значение правого выражения, если значение левого выражения `Неопределено`.

Эту операцию можно проиллюстрировать на предыдущем примере. Например, если адрес не задан, не нужно никого оповещать, а нужно просто написать, что данных нет.

```
структура Сотрудник
    пер ФИО: Строка
    пер Адрес: ПочтовыйАдрес?
    пер Телефон: Строка?
;

структура ПочтовыйАдрес
    пер Страна: Строка?
    пер Город: Строка
    пер Улица: Строка?
    пер Дом: Строка?
;

метод Скрипт()
    пер Менеджер = новый Сотрудник("Иванов")
    пер СтранаСотрудника = Менеджер.Адрес?.Страна ?? "Страна неизвестна"
;
```

В результате в переменной **СтранаСотрудника** будет значение **"Страна неизвестна"**.

Работа с числами, методы чисел

> **Примечание:**
>
> Работа с числами на базовом уровне описана здесь *(39)*.

Написать число в тексте скрипта

Числовое значение в тексте скрипта можно записать с помощью литерала или с помощью конструктора.

Литерал

Литерал позволяет задать десятичное, шестнадцатеричное или двоичное число.

Литерал десятичного числа содержит последовательность цифр, в качестве разделителя целой и дробной части используется точка, а разделитель разрядов не используется. Например:

```
метод Скрипт()
    пер Длина = 8165.27
;
```

Шестнадцатеричные числа могут использоваться, например, для задания цвета во внешних системах, с которыми взаимодействует ваш скрипт. Литерал шестнадцатеричного числа начинается с символов **0x**, за

которыми следуют шестнадцатеричные цифры, обозначаемые цифрами от 0 до 9 и латинскими буквами от A до F в верхнем или нижнем регистре. Например, в следующем примере задан красный цвет:

```
метод Скрипт()
    знч Цвет = 0xFF0000
;
```

Двоичные числа могут использоваться, например, при взаимодействии с внешними устройствами или при чтении данных в двоичных форматах. Литерал двоичного числа начинается с символов **0b**, за которыми следуют цифры 0 или 1. Например:

```
метод Скрипт()
    знч Маска = 0b10000111
;
```

Конструктор

С помощью конструктора можно создать число из его строкового представления. Поддерживаются строковые представления целых, дробных чисел и чисел в экспоненциальной записи.

Целые числа:

```
метод Скрипт()
    пер МоеЧисло = новый Число("123")
;
```

В результате в переменной **МоеЧисло** будет значение **123**.

Дробные числа:

```
метод Скрипт()
    пер МоеЧисло1 = новый Число("+1.23")
    пер МоеЧисло2 = новый Число("-12.3")
;
```

В результате:

- в переменной **МоеЧисло1** будет значение **1.23**;
- в переменной **МоеЧисло2** будет значение **-12.3**.

Экспоненциальная запись:

```
метод Скрипт()
    пер МоеЧисло1 = новый Число("1.23e+2")
    пер МоеЧисло2 = новый Число("-1.23e2")
    пер МоеЧисло3 = новый Число("123E-2")
;
```

В результате:

- в переменной **МоеЧисло1** будет значение **123**;
- в переменной **МоеЧисло2** будет значение **-123**;
- в переменной **МоеЧисло3** будет значение **1.23**.

Выделить целую часть числа

Выделение целой части может понадобиться, когда дробная часть числа не важна и просто откидывается. Например, нужно узнать расстояние в километрах, без учёта метров.

Для этого используйте метод `Число.ЦелаяЧасть()`:

```
метод Скрипт()
    знч Длина = 12345
    пер Километров = (Длина / 1000).ЦелаяЧасть()
;
```

В результате в переменной **Километров** будет значение **12**.

Получить представление числа в привычном виде

Представление числа в привычном виде может понадобиться, когда нужно показать число на экране или напечатать его.

Для этого используйте метод `Число.Представление()`. Он вернёт обозначение числа, в котором:

- целая часть сгруппирована по три разряда;
- группы разрядов отделяются друг от друга неразрывным пробелом;
- целая часть отделяется от дробной с помощью запятой;
- количество разрядов дробной части не ограничено.

Например:

```
метод Скрипт()
    пер Сумма = 250000.532
    пер ПредставлениеЧисла = Сумма.Представление()
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"250 000,532"**.

Кроме этого в методе `Число.Представление()` можно использовать форматную строку, чтобы изменить стандартное представление числа. Об этом будет рассказано в следующих примерах.

Получить представление числа в финансовом виде

Представление числа в финансовом виде может понадобиться, когда число обозначает денежную сумму. Денежные суммы представляются в рублях, поэтому у них два разряда в дробной части. Кроме этого для удобства чтения больших сумм принято разделять целую часть пробелами по три разряда.

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"_.2"**. Здесь:

- символ подчёркивания «_» — это флаг, который указывает, что целая часть будет сгруппирована по три разряда, группы разрядов по умолчанию будут разделены неразрывным пробелом;
- символы «.2» — это точность, которая указывает, что в дробной части будет два разряда.

Например:

```
метод  Скрипт()
      пер  Сумма  =  250000.532
      пер  ПредставлениеЧисла  =  Сумма.Представление("_.2")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"250 000,53"**.

Если в исходном числе количество знаков больше двух, значение будет округляться в режиме `РежимОкругления.ПоловинаВверх`, о котором рассказывается в примере Получить представление числа с округлением *(162)*. Если в исходном числе количество знаков меньше двух, значение будет дополняться нулями справа.

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа для другой страны

Представление числа для другой страны может понадобиться для формирования документов или экранных сообщений, предназначенных иностранным пользователям. Например, в Китае принято разделять группы разрядов запятой «,», а дробную часть числа отделять от целой точкой «.».

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"_; РГ=’; РД=."**. Здесь:

- символ подчёркивания «_» — это флаг, который указывает, что целая часть будет сгруппирована по три разряда;
- символы «РГ=,» — это специальный флаг, который указывает, что разделителем групп разрядов будет запятая «,»;
- символы «РД=.» — это специальный флаг, который указывает, что разделителем целой и дробной части будет точка «.».

Например:

```
метод  Скрипт()
      пер  Сумма  =  82457.538
      пер  ПредставлениеЧисла  =  Сумма.Представление("_;  РГ=,;  РД=.")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"82,457.538"**.

Флаг `РГ` применяется только вместе с флагом «_», включающим разделение по группам. Специальные флаги отделяются друг от друга и от других составляющих форматной строки точкой с запятой «;».

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа, дополненное нулями слева

Представление числа, дополненное нулями слева, может понадобиться для формирования последовательных кодов или номеров документов, которые имеют тип `Строка` и одинаковую длину.

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"08"**. Здесь:

- символ «0» — это флаг, который указывает, что свободное место слева заполняется нулями;
- символ «8» — это ширина, которая указывает максимальное количество символов в том представлении, которое будет получено.

Например:

```
метод  Скрипт()
      пер  Счетчик  =  253
      пер  НомерДокумента  =  Счетчик.Представление("08")
;
```

В результате в переменной **НомерДокумента** будет значение **"00000253"**.

Чтобы лидирующие нули слева стали видны, нужно, чтобы общая ширина выводимого числа (в этом примере 8) была больше, чем количество знаков в дробной и целой части, включая разделитель (в этом примере 3). Тогда оставшееся место дополняется нулями (5 = 8 - 3). Если общая ширина будет меньше или равна ширине выводимого числа либо не указана, то нули слева выводиться не будут.

Примечание:

Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа со знаком «+»

Представление числа с плюсом «+» может понадобиться для явного выделения сумм, которые являются приращением («увеличилось на ...»).

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"+"**. Здесь плюс «+» — это флаг, который указывает, что положительные числа выводятся с плюсом «+».

Например:

```
метод  Скрипт()
      пер  Сумма  =  253.58
      пер  ПредставлениеЧисла  =  Сумма.Представление("+")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"+253,58"**.

Примечание:

Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа с округлением

Представление числа с округлением может понадобиться для указания величин с принятой точностью, когда более точные значения либо не интересны, либо недостоверны.

Например, на круговой диаграмме нужно показать числовые значения каждой точки в процентах. Как правило, для этого используют либо целые числа, либо числа с точностью до одного разряда. Более точные значения не нужны для визуализации.

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **".1; ОК=ПВВ"**. Здесь:

- символы «.1» — это точность, которая указывает, что в дробной части будет один разряд;
- символы «ОК=ПВВ» — это специальный флаг, который указывает, что значение отбрасываемого разряда, больше или равное 5, будет округлено в большую сторону.

Например:

```
метод Скрипт()
    пер ДоляВПроцентах = 53.57
    пер ПредставлениеВПроцентах = ДоляВПроцентах.Представление(".1; ОК=ПВВ")
;
```

В результате в переменной **ПредставлениеВПроцентах** будет значение **"53,6"**.

Специальный флаг **ОК** может иметь и другие значения. Например:

- **ПВН** — ПоловинаВниз. Если отбрасываемый разряд < 5, округляет в меньшую сторону. Например:

  ```
  метод Скрипт()
      пер ДоляВПроцентах = 53.54
      пер ПредставлениеВПроцентах = ДоляВПроцентах.Представление(".1; ОК=ПВН")
  ;
  ```

  В результате в переменной **ПредставлениеВПроцентах** будет значение **"53,5"**.

- **ВВ** — Вверх. Округляет всегда в большую сторону. Например:

  ```
  метод Скрипт()
      пер ДоляВПроцентах = 53.54
      пер ПредставлениеВПроцентах = ДоляВПроцентах.Представление(".1; ОК=ВВ")
  ;
  ```

  В результате в переменной **ПредставлениеВПроцентах** будет значение **"53,6"**.

- **ВН** — Вниз. Округляет всегда в меньшую сторону. Например:

  ```
  метод Скрипт()
      пер ДоляВПроцентах = 53.57
      пер ПредставлениеВПроцентах = ДоляВПроцентах.Представление(".1; ОК=ВН")
  ;
  ```

  В результате в переменной **ПредставлениеВПроцентах** будет значение **"53,5"**.

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Округлить число

Округление числа может понадобиться для получения значений с принятой точностью, когда более точные значения не имеют смысла.

Например, нужно установить цену на товар исходя из фиксированной прибыли. Для этого планируемую выручку нужно разделить на количество товара. Результат такого деления может иметь пять или шесть разрядов дробной части. Но цена на товар не может быть точнее двух разрядов, минимальной денежной единицей является копейка. Поэтому в таком случае используется округление в большую сторону.

Для этого используйте метод `Число.Округлить()` с указанием разрядности и режима округления. Например:

```
метод Скрипт()
    пер Сумма = 253.57
    пер ЗначениеЧисла = Сумма.Округлить(1, РежимОкругления.Вниз)
;
```

В результате в переменной **ЗначениеЧисла** будет значение **253.5**.

Здесь:

- первый аргумент, **1**, — это разрядность, до которой нужно округлить;
- второй аргумент, **РежимОкругления.Вниз**, — это режим округления, который нужно использовать.

Если первый аргумент равен нулю или вообще пропущен, то значение числа будет округлено до целого. Например:

```
метод Скрипт()
    пер Сумма = 253.57
    пер ЗначениеЧисла = Сумма.Округлить(РежимОкругления.ПоловинаВверх)
;
```

В результате в переменной **ЗначениеЧисла** будет значение **254**.

Если первый аргумент отрицательный, то число будет округлено до соответствующего разряда в целой части, начиная с младших разрядов. Например:

```
метод Скрипт()
    пер Сумма = 253.57
    пер ЗначениеЧисла = Сумма.Округлить(-1, РежимОкругления.Вверх)
;
```

В результате в переменной **ЗначениеЧисла** будет значение **260**.

Получить представление числа с указанием валюты

Представление числа с указанием валюты может понадобиться для формирования финансовых документов.

Например, если валютой является рубль, то знак рубля «₽» пишется через пробел после числа — 135,00 ₽. Если валютой является евро, то знак евро «€» пишется перед числом и тоже с пробелом (€ 621,00).

Чтобы указать валюту евро используйте метод `Число.Представление()` с указанием следующей форматной строки: **"'€ '.2"**. Здесь:

- символы «'€ '» — это префикс, он содержит текст, заключённый в апострофы «'», который будет выводиться перед числом;
- символы «.2» — это точность, которая указывает, что в дробной части будет два разряда.

Например:

```
метод Скрипт()
    пер Сумма = 123.456
    пер ПредставлениеЧисла = Сумма.Представление("'€ '.2")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"€ 123,46"**.

Чтобы указать валюту рубль, используйте метод `Число.Представление()` с указанием следующей форматной строки: **".2' ₽'"**. Здесь:

- символы «.2» — это точность, которая указывает, что в дробной части будет два разряда;
- символы «' ₽'» — это суффикс, он содержит текст, заключённый в апострофы «'», который будет выводиться после числа.

Например:

```
метод Скрипт()
    пер Сумма = 123.71
    пер ПредставлениеЧисла = Сумма.Представление(".2' ₽'")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"123,71 ₽"**.

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа в виде процента

Представление числа в виде процента может понадобиться для вывода процентных соотношений.

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"Процент"**. Здесь `Процент` — это именованный стандартный формат, который указывает, что:

1. Число умножается на 100.
2. Выводится в общепринятом финансовом виде:
    - целая часть группируется по три разряда;
    - группы разрядов разделяются неразрывным пробелом;
    - дробная часть отделяется от целой запятой.
3. После числа пишется процент «%».

Например:

```
метод Скрипт()
    пер Отношение = 0.123456
    пер ПредставлениеЧисла = Отношение.Представление("Процент")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"12,35%"**.

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

Получить представление числа в произвольном денежном виде

Представление числа в произвольном денежном виде может понадобиться для вывода денежных сумм в свободной форме с указанием валюты.

Например, в договоре надо написать: «… рублей» или «… долларов США».

Для этого используйте метод `Число.Представление()` с указанием следующей форматной строки: **"Деньги"**. Здесь `Деньги` — это именованный стандартный формат, который указывает, что число выводится в общепринятом финансовом виде:

- целая часть группируется по три разряда;
- группы разрядов разделяются неразрывным пробелом;
- дробная часть отделяется от целой запятой;

Например:

```
метод Скрипт()
    пер Сумма = 1237.423
    пер ПредставлениеЧисла = Сумма.Представление("Деньги")
;
```

В результате в переменной **ПредставлениеЧисла** будет значение **"1 237,42"**.

> **Примечание:**
>
> Подробнее о правилах написания форматной строки читайте в справке по стандартной библиотеке.

> **Примечание:**
>
> Подробнее о типе `Число` читайте в справке по стандартной библиотеке.

Работа со строками, методы строк

> **Примечание:**
>
> Работа со строками на базовом уровне описана здесь *(42)*.

Неявное преобразование к типу «Строка» при конкатенации

Порой к строке нужно добавить часть, которая строкой не является. Например, вы скачиваете с сервера много файлов. Эта операция выполняется долго, и, чтобы не скучать, вы выводите на экран надпись «Скачано 10%» . Число 10 здесь для примера. На самом деле вы знаете общее количество файлов, которое надо скачать, и каждый раз, как вы скачали очередной файл, вы подсчитываете, сколько процентов скачано, и меняете надпись.

По-правильному, число, обозначающее количество процентов, нужно сначала преобразовать к типу `Строка`, а после этого с помощью конкатенации «склеить» строку из трёх частей: «Скачано », «10» и «%».

Однако лучше и эффективнее воспользоваться возможностью неявного преобразования к типу `Строка` при конкатенации. Если в операции конкатенации первый операнд имеет тип `Строка`, то все следующие за ним операнды «1С:Элемент» автоматически преобразует к типу `Строка` и выполняет конкатенацию. То есть не нужно заранее самому преобразовывать операнд к типу `Строка`.

Например:

```
метод Скрипт()
    пер СколькоПроцентов = 10
    пер Сообщение = "Скачано " + СколькоПроцентов + "%"
;
```

В результате в переменной **Сообщение** будет значение **"Скачано 10%"**.

Массовая конкатенация

Массовая конкатенация — это «склеивание» 1000 и более строк. 1000 — приблизительное количество. Если строки короткие, их может быть больше 1000. Если строки длинные, их может быть меньше 1000: чем строки длиннее, тем операции выполняются дольше.

В любом случае, когда нужно «склеить» много строк, используйте не конкатенацию, а метод `Строки.Соединить()`. Такой код не только быстрее выполняется, но и потребляет меньше оперативной памяти.

Метод `Строки.Соединить()` получает массив строк и возвращает одну строку, «склеенную» из всех элементов массива.

Например:

```
метод Скрипт()
	пер СтихотворныеСтроки = новый Массив<Строка>()
	СтихотворныеСтроки.Добавить("У лукоморья дуб зелёный;")
	СтихотворныеСтроки.Добавить("Златая цепь на дубе том:")
	СтихотворныеСтроки.Добавить("И днём и ночью кот учёный")
	СтихотворныеСтроки.Добавить("Всё ходит по цепи кругом;")
	СтихотворныеСтроки.Добавить("Идёт направо - песнь заводит,")
	СтихотворныеСтроки.Добавить("Налево - сказку говорит.")
	СтихотворныеСтроки.Добавить("Там чудеса: там леший бродит,")
	СтихотворныеСтроки.Добавить("Русалка на ветвях сидит;")
	пер Стихотворение: Строка = Строки.Соединить(СтихотворныеСтроки, Символы.НОВАЯ_СТРОКА)
	Консоль.Записать(Стихотворение)
;
```

В результате в переменной **Стихотворение** будет следующее значение:

```
У лукоморья дуб зелёный;
Златая цепь на дубе том:
И днём и ночью кот учёный
Всё ходит по цепи кругом;
Идёт направо - песнь заводит,
Налево - сказку говорит.
Там чудеса: там леший бродит,
Русалка на ветвях сидит;
```

Интерполяция

Кроме конкатенации можно использовать интерполяцию строк. Во многих случаях она может выглядеть более понятно.

Интерполяция строк — это процесс вычисления значения строкового литерала, который содержит одно или несколько выражений интерполяции.

Выражение интерполяции это, грубо говоря, имя переменной, значение которой нужно подставить в литерал. Только перед именем есть специальный символ, который указывает, каким именно образом нужно получить строковое значение этой переменной:

- %**имя_переменной** — для преобразования значения используется метод, унаследованный от `Объект.ВСтроку()`;
- $**имя_переменной** — для преобразования значения используется метод, унаследованный от `Объект.Представление()`.

> **Примечание:**
>
> Знак доллара «$» можно вводить с помощью Alt-ввода *(227)*.

Для большинства прикладных задач вам подойдёт вариант со знаком доллара «$». Преобразуя значение к типу `Строка`, он учитывает региональные настройки компьютера и другие настройки. Поэтому, например, преобразуя значение типа `Дата`, вы получите его в привычном вам виде.

Посмотрите, в чём отличие интерполяции от конкатенации и почему интерполяция может выглядеть удобнее.

В примере со скачиванием файлов *(166)* вы писали так:

```
метод Скрипт()
    пер СколькоПроцентов = 10
    пер Сообщение = "Скачано " + СколькоПроцентов + "%"
;
```

С использованием интерполяции этот пример будет выглядеть так:

```
метод Скрипт()
    пер СколькоПроцентов = 10
    пер Сообщение = "Скачано $СколькоПроцентов%"
;
```

В результате в переменной **Сообщение** будет значение **"Скачано 10%"**.

Прямо в то место, где должно находиться **10**, вы вставляете выражение интерполяции — **$СколькоПроцентов**.

На самом деле выражение интерполяции может содержать не просто имя переменной, а целое выражение, которое записывается в фигурных скобках. Например, **${Длина * Ширина}**:

```
метод Скрипт()
    пер Длина = 100
    пер Ширина = 30
    пер Площадь = "Площадь равна ${Длина * Ширина} м2"
;
```

В результате в переменной **Площадь** будет значение **"Площадь равна 3 000 м2"**.

> **Примечание:**
>
> Фигурные скобки «{» «}» можно вводить с помощью Alt-ввода *(227)*.

Можно указать правила форматирования, в соответствии с которыми будет обработано вставляемое значение. Эта возможность доступна для тех выражений, результат вычисления которых имеет тип, унаследованный от типа `Форматируемое`.

Для этого после выражения укажите вертикальную черту «|» и форматную строку. В этом случае используется синтаксис выражения интерполяции со знаком доллара «${}». Для преобразования значения неявно используется метод, унаследованный от `Объект.Представление()`.

Например:

```
метод Скрипт()
    пер Сообщение = "Текущая дата: ${ДатаВремя.Сейчас()|дд ММММ гггг, дддд}"
;
```

В результате в переменной **Сообщение** будет, например, значение **"Текущая дата: 12 июня 2025, четверг"**.

> **Примечание:**
>
> Вертикальную черту «|» можно вводить с помощью Alt-ввода *(227)*.

Описание форматной строки можно посмотреть для каждого типа отдельно. Обычно форматная строка описана в справке по стандартной библиотеке, в методе, унаследованном от `Объект.Представление()`. Например:

- Число.Представление();
- ДатаВремя.Представление() и т. д.

> **Примечание:**
>
> Подробнее про интерполяцию читайте в руководстве разработчика.

Индекс

Строка — это последовательность символов. При работе со строками возникают задачи, когда нужно либо прочитать какой-то символ строки, либо вставить в строку что-то в определённую позицию. В таких задачах используется индекс символа. *Индекс символа* — это число, порядковый номер символа в строке. Индекс первого символа всегда 0.

Например, в строке **Пример** символ **П** имеет индекс **0**, а символ **и** имеет индекс **2**.

Написать строку в тексте скрипта

Значение типа `Строка` в тексте скрипта можно записать с помощью литерала. Литерал строки содержит произвольный набор символов, заключённый в двойные кавычки. Например:

```
метод Скрипт()
    пер Сообщение = "Привет! Это мой номер телефона: +7(495) 688-90-02"
;
```

Если объявить переменную типа `Строка` без указания значения инициализации, то значением этой переменной по умолчанию будет пустая строка (""). Например:

```
метод Скрипт()
    пер Сообщение: Строка
;
```

Значение типа `Строка` может состоять из нескольких строк или содержать символы табуляции. Для обозначения конца строк и табуляции используются некоторые обычные символы, предварённые обратной косой чертой «\»:

- \н — символ новой строки LF;
- \в — символ возврата каретки CR;
- \т — символ табуляции TAB.

Например, вы хотите вывести многострочное сообщение, разбитое на короткие понятные фрагменты. Для этого вставьте в текст сообщения в местах разбиения символы **\в\н**. Например:

```
метод Скрипт()
    пер Сообщение = "Внимание!\в\нВы не заполнили поля:\в\н- номер телефона\в\н- электронная почта"
;
```

В результате в переменной **Сообщение** будет следующее значение:

```
Внимание!
Вы не заполнили поля:
- номер телефона
- электронная почта
```

Некоторые символы в строке являются управляющими, они выполняют служебные задачи и сами по себе не выводятся. Это двойная кавычка «"», обратная косая черта «\», процент «%» и знак доллара «$». Как их вывести, рассказано в примере Добавить в строку специальные символы *(170)*.

Добавить в строку специальные символы

Некоторые символы в строке являются управляющими, они выполняют служебные задачи и сами по себе не выводятся. Это символы:

- Двойная кавычка «"» — обозначает начало и конец строкового литерала.
- Процент «%» и знак доллара «$» — обозначают переменную интерполяции или начало выражения интерполяции *(167)*.
- Обратная косая черта «\» — обозначает, что следующий за ним символ используется не в своём обычном значении.

Чтобы вывести эти символы, перед ними нужно добавить другой управляющий символ — обратную косую черту «\». Это называется *экранированием*. Например:

```
метод Скрипт()
    пер Фраза = "Язык \"1С:Элемента\""
    пер Скидка = "Скидка клиента: 20\%"
    пер Цена = "Цена товара: \$ 55"
;
```

В результате:

- В переменной **Фраза** будет значение **"Язык "1С:Элемента""**.
- В переменной **Скидка** будет значение **"Скидка клиента: 20%"**.
- В переменной **Цена** будет значение **"Цена товара: $ 55"**.

Чтобы использовать в строке обратную косую черту, её нужно экранировать самой собой. Например:

```
метод Скрипт()
    пер РабочийКаталог = "C:\\test\\"
;
```

В результате в переменной **РабочийКаталог** будет значение **«C:\test\»**.

Также последовательность символов «\ю» позволяет получить любой символ, доступный в кодировке Unicode. Обозначение нужно записать как: **"\юXXXXX"**, где XXXXX — это десятичный код символа. Если знаков меньше пяти, их нужно дополнить лидирующими нулями. Например:

```
метод Скрипт()
    пер Символ = "символ \ю00064-собака"
;
```

В результате в переменной **Символ** будет значение **"символ @-собака"**.

Будьте внимательны! Все управляющие последовательности, содержащиеся в литерале строки (\н, \в, \т, \юXXXXX, \%, \$, \"), интерпретируются как один символ. Это надо учитывать при посимвольном доступе к таким строкам, при работе с методами `Строка.Длина()`, `Строка.Подстрока()`, `Строка.ЗаменитьДиапазон()` и им подобными.

Написать многострочную строку в тексте скрипта

Многострочная строка может понадобиться для записи в тексте скрипта фрагментов текстовых файлов (.html-, .xml-, .css-файлов и т. п.).

Для этого используйте многострочный литерал. Многострочные литералы полностью сохраняют форматирование по открывающей кавычке, которая обязательно должна находиться на новой строке. Символ переноса строки указывать не нужно. Например:

```
метод Скрипт()
   знч СтрокаXml =
      "<note>
         <heading>Описание объектной модели</heading>
         <body>В основании всей иерархии типов лежит тип Объект.</body>
      </note>"
;
```

В результате в переменной **СтрокаXml** будет следующее значение:

```
<note>
    <heading>Описание объектной модели</heading>
    <body>В основании всей иерархии типов лежит тип Объект.</body>
</note>
```

Последовательно перебрать символы в строке

Последовательный перебор символов в строке может понадобиться, чтобы найти первую цифру или первую букву в строке.

Например, имя файла **префикс1357** состоит из префикса **префикс** и номера **1357**. Новому файлу надо дать имя с тем же префиксом и номером, увеличенным на единицу — **префикс1358**.

Чтобы выделить префикс и номер, нужно знать позицию первой цифры в строке. Для этого нужно последовательно перебрать символы, начиная с начала, и проверить, является ли символ цифрой (см. пример Узнать, содержатся ли в исходной строке только буквы, только цифры или только буквы и цифры *(180)*).

Чтобы последовательно перебрать символы, используйте операцию доступа по индексу *(169)* **[ ]** или эквивалентный ей метод `Строка.Символ()`:

```
метод Скрипт()
    пер ИмяСтарогоФайла = "префикс1357"
    для Счетчик = 0 по ИмяСтарогоФайла.Длина() - 1
        если ИмяСтарогоФайла[Счетчик].ТолькоЦифры()
            // Когда Счетчик = 7 — выделить из строки префикс и номер.
        ;
    ;
;
```

Как выделить из строки префикс и номер, смотрите в примере Получить часть строки *(172)*.

Как преобразовать строку в число и обратно, смотрите в примерах Получить число из строки *(173)* и Интерполяция *(167)*.

Получить часть строки

Получение части строки может понадобиться для выделения префикса и номера из имени файла.

Пусть имя файла — **префикс1357**. Известно, что первая цифра в имени имеет индекс *(169)* 7. Нужно выделить префикс (**префикс**) и номер (**1357**).

Для этого используйте методы `Строка.ПодстрокаСНачала()` и `Строка.Подстрока()`:

```
метод Скрипт()
    пер ИмяСтарогоФайла = "префикс1357"
    пер ИндексПервойЦифры = 7
    пер Префикс = ИмяСтарогоФайла.ПодстрокаСНачала(ИндексПервойЦифры)
    пер Номер = ИмяСтарогоФайла.Подстрока(ИндексПервойЦифры)
;
```

В результате:

- В переменной **Префикс** будет значение **"префикс"**.
- В переменной **Номер** будет значение **"1357"**.

Здесь:

- В методе `Строка.ПодстрокаСНачала()` указан единственный аргумент — количество символов, которые надо получить.
- В методе `Строка.Подстрока()` указан только первый аргумент — индекс символа, начиная с которого нужно получить подстроку. Второй аргумент не указан, это значит, что будет получена подстрока до конца строки.

Чтобы получить часть строки с конца, используйте метод `Строка.ПодстрокаСКонца()`.

Как узнать позицию первой цифры в строке, смотрите в примере Последовательно перебрать символы в строке *(171)*.

Найти вхождение подстроки

Поиск вхождения подстроки может понадобиться для выделения префикса и номера из номера документа.

Например, известно, что номер документа, **Договор_003**, состоит из префикса и номера, которые отделены друг от друга символом подчёркивания «_» . Чтобы выделить из строки префикс (**Договор**) и номер (**003**), нужно сначала найти позицию символа-разделителя в исходной строке.

Для этого используйте метод `Строка.Найти()`:

```
метод  Скрипт()
      пер  НомерДокумента  =  "Договор_003"
      пер  ИндексРазделителя  =  НомерДокумента.Найти("_")
;
```

В результате в переменной **ИндексРазделителя** будет значение **7**.

Здесь в методе `Строка.Найти()`:

- Указан только первый аргумент — подстрока, которую нужно найти.
- Второй и третий аргументы не указаны, это значит, что поиск будет выполняться во всей строке.

Чтобы найти последнее вхождение разделителя в строке, используйте метод `Строка.НайтиСКонца()`.

Как получить часть строки, смотрите в примере Получить часть строки *(172)*.

Получить число из строки

Получение числа из строки может понадобиться для того, чтобы номер, полученный из имени файла, увеличить на единицу.

Для этого используйте конструктор типа `Число` (**новый Число()**):

```
метод  Скрипт()
      пер  СтарыйНомер  =  "1357"
      пер  НовыйНомер  =  новый  Число(СтарыйНомер)  +  1
;
```

В результате в переменной **НовыйНомер** будет значение **1358**.

Аргумент конструктора должен быть записан в форме десятичного литерала числа. Как выглядит десятичный литерал числа, смотрите в примере Написать число в тексте скрипта *(157)*.

Проверить, что строка заполнена

Проверка строки на заполненность может понадобиться для контроля заполнения обязательных полей.

Для этого используйте метод `Строка.Пусто()`.

```
метод Скрипт()
    пер Поле: Строка
    если Поле.Пусто()
        // Сообщить, что поле должно быть заполнено.
    ;
;
```

Строковая переменная, для которой не задано значение, имеет стандартное значение «пустая строка» «"»». Метод `Строка.Пусто()` проверяет, что строка не содержит символы.

Вставить одну строку в другую

Вставка одной строки в другую может понадобиться для помещения символов в заданное место строки. Например, номер банковской карты для удобства представления пользователю нужно разбить на группы по четыре цифры и вставить между ними дефис «-».

Для этого используйте метод `Строка.Вставить()`:

```
метод Скрипт()
    пер НомерКарты = "2222333344445555"

    // Можно использовать текучий интерфейс, т.к. метод возвращает строку, полученную в результате вставки.
    НомерКарты = НомерКарты.Вставить(4, "-").Вставить(9, "-").Вставить(14, "-")
;
```

В результате в переменной **НомерКарты** будет значение **"2222-3333-4444-5555"**.

Здесь в методе `Строка.Вставить()`:

- В первом аргументе задаётся индекс, в который нужно вставить подстроку.
- Во втором аргументе задаётся подстрока, которую нужно вставить.

Удалить вхождения подстроки

Удаление подстроки может понадобиться для удаления части путей к картинкам при обработке .html-файлов, когда картинки переносятся в другой каталог.

Например, есть .html-файл, содержащий следующий фрагмент:

```
<body>
    <p>Картинка 1 <img src="C:\Example\Images\pic1.png" /></p>
    <p>Картинка 2 <img src="C:\Example\Images\pic2.png" /></p>
    <p>Картинка 3 <img src="C:\Example\Images\pic3.png" /></p>
</body>
```

Этот фрагмент находится в переменной **ТекстФайла**. Все картинки находятся в каталоге **C:\Example\Images**. Предположим, вы переместили картинки в другой каталог **C:\Images**. Чтобы документ правильно их показывал, во всех путях к картинкам нужно удалить **\Example**.

Для этого используйте метод `Строка.Удалить()`:

```
метод Скрипт()
    пер ТекстФайла =
        "<body>
            <p>Картинка 1 <img src=\"C:\\Example\\Images\\pic1.png\" /></p>
            <p>Картинка 2 <img src=\"C:\\Example\\Images\\pic2.png\" /></p>
            <p>Картинка 3 <img src=\"C:\\Example\\Images\\pic3.png\" /></p>
        </body>"
    пер НовыйТекст = ТекстФайла.Удалить("\Example")
;
```

В результате переменная **НовыйТекст** будет иметь следующее значение:

```
<body>
    <p>Картинка 1 <img src="C:\Images\pic1.png" /></p>
    <p>Картинка 2 <img src="C:\Images\pic2.png" /></p>
    <p>Картинка 3 <img src="C:\Images\pic3.png" /></p>
</body>
```

Здесь в методе `Строка.Удалить()`:

- Указан только первый аргумент — подстрока, которую нужно удалить.
- Второй аргумент не указан, это значит, что будут удалены все вхождения этой подстроки.

Удалить диапазон символов

Иногда бывает нужно удалить не подстроку, а конкретный диапазон символов — например, первые два символа в номере года документа.

Например, документы стандартно пронумерованы так, что сначала идёт префикс, потом год и потом номер документа — **Договор_2022-203**. Задача состоит в том, чтобы уменьшить длину номеров и удалить из них первые два символа года, так как они во всех документах одинаковые.

Для этого используйте метод `Строка.УдалитьДиапазон()`:

```
метод Скрипт()
    пер СтарыйНомер = "Договор_2022-203"
    пер НовыйНомер = СтарыйНомер.УдалитьДиапазон(8, 10)
;
```

В результате в переменной **НовыйНомер** будет значение **"Договор_22-203"**.

Здесь в методе `Строка.УдалитьДиапазон()`:

- В первом аргументе задаётся индекс символа, с которого начинается удаляемая подстрока.
- Во втором аргументе задаётся индекс первого символа, который следует за удаляемой подстрокой.

Заменить вхождения подстроки в исходной строке

Замена вхождения подстроки может понадобиться при обработке .html-файлов, когда картинки преобразуются в единый формат и в тексте нужно заменить расширение файлов картинок.

Например, есть .html-файл, содержащий следующий фрагмент:

```
<body>
    <p>Картинка 1 <img src="C:\Example\Images\pic1.jpg" /></p>
    <p>Картинка 2 <img src="C:\Example\Images\pic2.jpg" /></p>
    <p>Картинка 3 <img src="C:\Example\Images\pic3.jpg" /></p>
</body>
```

Фрагмент .html-файла находится в переменной **ТекстФайла**. Нужно заменить расширения всех файлов картинок, вместо **jpg** сделать **png**.

Для этого используйте метод `Строка.Заменить()`:

```
метод Скрипт()
    пер ТекстФайла =
        "<body>
            <p>Картинка 1 <img src=\"C:\\Example\\Images\\pic1.png\" /></p>
            <p>Картинка 2 <img src=\"C:\\Example\\Images\\pic2.png\" /></p>
            <p>Картинка 3 <img src=\"C:\\Example\\Images\\pic3.png\" /></p>
        </body>"
    пер НовыйТекст = ТекстФайла.Заменить("jpg", "png")
;
```

В результате переменная **НовыйТекст** будет иметь следующее значение:

```
<body>
    <p>Картинка 1 <img src="C:\Example\Images\pic1.png" /></p>
    <p>Картинка 2 <img src="C:\Example\Images\pic2.png" /></p>
    <p>Картинка 3 <img src="C:\Example\Images\pic3.png" /></p>
</body>
```

Здесь в методе `Строка.Заменить()`:

- Указаны только первый и второй аргументы — подстрока, которую нужно заменить, и подстрока замены.
- Третий аргумент не указан, это значит, что будут заменены все вхождения подстроки, заданной в первом аргументе.

Получить массив подстрок из исходной строки

Получение массива подстрок может понадобиться для того, чтобы разделить на отдельные строки текст, прочитанный из текстового файла.

В разных операционных системах используются разные символы окончания строк: в Windows — «\в \н»; в Linux — «\н»; в macOS — «\в». Поэтому если заранее неизвестно, в формате какой операционной системы получен текстовый файл, то для разделения строки на части по символам окончания строк удобно использовать метод `Строка.ПолучитьСтроки()`.

Например, есть лог-файл, содержащий следующий фрагмент:

```
2025/08/12-15:01:09.406,pid=15228,tid=6072,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=6072,component=launcher,class=JMethodCaller…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=JavaVmManager…
```

Фрагмент находится в переменной **Лог**. Чтобы разделить этот текст на отдельные строки, используйте метод `Строка.ПолучитьСтроки()`:

```
метод Скрипт()
    пер Лог =
        "2025/08/12-15:01:09.406,pid=15228,tid=6072,component=launcher,class=Daemon…
        2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
        2025/08/12-15:01:09.422,pid=15228,tid=6072,component=launcher,class=JMethodCaller…
        2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
        2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=JavaVmManager… "
    для СтрокаЛога из Лог.ПолучитьСтроки()
        // Обработать одну строку СтрокаЛога
    ;
;
```

В результате вы получите массив строк, которые можно обойти и проанализировать в цикле.

Чтобы соединить подстроки в строку с разделителем, используйте метод `Строки.Соединить()`.

Разделить строку на части по символам-разделителям

Разделение строки на части по разделителям может понадобиться для обработки .csv-файлов. CSV — это текстовый формат представления табличных данных. В таком файле строка текста соответствует строке таблицы, а столбцы отделены один от другого символом-разделителем. В качестве такого символа может использоваться запятая, точка с запятой, табуляция и другие символы. В формат CSV может сохранять данные Microsoft Excel. Также этот формат часто используют для журналов и логов.

Например, есть лог-файл, в котором параметры отделены друг от друга запятой «,»:

```
2025/08/12-15:01:09.406,pid=15228,tid=6072,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=6072,component=launcher,class=JMethodCaller…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=JavaVmManager…
```

Содержимое этого файла находится в переменной **Лог**. Чтобы разделить каждую строку на составляющие, используйте метод `Строка.Разделить()`:

```
метод Скрипт()
    пер Лог =
    "2025/08/12-15:01:09.406,pid=15228,tid=6072,component=launcher,class=Daemon…
    2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
    2025/08/12-15:01:09.422,pid=15228,tid=6072,component=launcher,class=JMethodCaller…
    2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=Daemon…
    2025/08/12-15:01:09.422,pid=15228,tid=18188,component=launcher,class=JavaVmManager… "
    для СтрокаЛога из Лог.ПолучитьСтроки()
        для ЧастьСтроки из СтрокаЛога.Разделить(",")
```

```
            // Обработать ЧастьСтроки
        ;
    ;
;
```

Сначала из файла получается массив строк, который потом обходится в цикле. Для деления файла на строки используется метод `Строка.ПолучитьСтроки()`, который описан в примере Получить массив подстрок из исходной строки *(176)*.

Здесь в методе `Строка.Разделить()`:

- Указан только первый аргумент — подстрока-разделитель.
- Второй аргумент не указан, это значит, что в результат не будут включены пустые подстроки, полученные в результате того, что два символа-разделителя шли подряд.

Сравнить строку с эталоном, игнорируя регистр символов

Сравнение строки с эталоном может понадобиться при анализе xml- или html-текстов, когда нужно находить теги с определёнными именами, а теги могут быть записаны как в верхнем, так и в нижнем регистре. Например, нужно найти тег **body**.

Для этого используйте метод `Строка.Сравнить()`:

```
метод Скрипт()
    пер ИмяТегаВФайле = "BODY"
    если ИмяТегаВФайле.Сравнить("body", Истина) == 0
        // Обработать тег <body>
    ;
;
```

Здесь в методе `Строка.Сравнить()`:

- В первом аргументе задаётся строка, с которой нужно сравнить исходную строку.
- Во втором аргументе задаётся признак того, что нужно игнорировать регистр сравниваемых строк.

Избавиться от дубликатов строк, отличающихся только пробелами в начале или в конце строки

Избавление от дубликатов строк может понадобиться при заполнении базы товаров. В ней не должно быть двух наименований, которые отличаются только случайно добавленными пробелами в начале или в конце. Чтобы сравнить новое наименование с теми, которые уже есть в базе, сначала нужно удалить пробельные символы, которые могут быть в начале и в конце наименования.

Для этого используйте метод `Строка.Сократить()`:

```
метод Скрипт()
    пер Наименование = " Товар "
    пер НовоеНаименование = Наименование.Сократить()
;
```

В результате в переменной **НовоеНаименование** будет значение **"Товар"**.

Чтобы удалить пробелы только в начале или только в конце строки, используйте методы `Строка.СократитьСНачала()` и `Строка.СократитьСКонца()`.

Узнать, начинается (заканчивается) ли исходная строка на заданную подстроку

Информация о том, что строка начинается с определённой подстроки, может понадобиться, чтобы определить, является гиперссылка внутренней ссылкой «1С:Предприятия» или нет. Внутренние ссылки «1С:Предприятия» начинаются с подстроки **e1c://**.

Для этого используйте метод `Строка.НачинаетсяС()`:

```
метод Скрипт()
    пер URL = "e1c://?WS=https://test.app-site.com/app-name"
    если URL.НачинаетсяС("e1c://")
        // Обработать ссылку 1С
    ;
;
```

Здесь в методе `Строка.НачинаетсяС()`:

- Указан только первый аргумент — подстрока, с которой должна начинаться исходная строка.
- Второй аргумент не указан, это значит, что при сравнении будет учитываться регистр строк.

Чтобы узнать, что строка заканчивается на заданную подстроку, используйте метод `Строка.ЗаканчиваетсяНа()`.

Дополнить строку символами с начала или с конца

Дополнение строки символами может понадобиться для того, чтобы имена файлов, пронумерованных по порядку, преобразовать к одинаковой длине, дополняя их нулями с начала.

Для этого используйте метод `Строка.ДополнитьСНачала()`:

```
метод Скрипт()
    пер ПервыйФайл = "2.txt"
    пер НовоеИмяПервогоФайла = ПервыйФайл.ДополнитьСНачала(7, "0")

    пер ВторойФайл = "43.txt"
    пер НовоеИмяВторогоФайла = ВторойФайл.ДополнитьСНачала(7, "0")
;
```

В результате:

- в переменной **НовоеИмяПервогоФайла** будет значение **"002.txt"**;
- в переменной **НовоеИмяВторогоФайла** будет значение **"043.txt"**.

Здесь:

- В первом аргументе задаётся длина строки, которая должна получиться в результате.
- Во втором аргументе задаётся символ, которым будет дополняться исходная строка.

Чтобы дополнить строку символами с конца, используйте метод `Строка.ДополнитьСКонца()`.

Узнать, содержится ли подстрока в исходной строке

Это может понадобиться для того, чтобы определённым образом обработать файлы в зависимости от того, какой год содержится в их имени — **log_2022-10-31.txt**.

Для этого используйте метод `Строка.Содержит()`:

```
метод Скрипт()
    пер ИмяФайла = "log_2022-10-31.txt"
    если ИмяФайла.Содержит("2010") или ИмяФайла.Содержит("2011")
        // Поместить файл в архив
    иначе если ИмяФайла.Содержит("2022")
        // Скопировать файл
    иначе
        // Удалить файл
    ;
;
```

В результате файлы 2010 и 2011 годов будут помещены в архив. Файлы 2022 года будут скопированы, а все остальные файлы будут удалены.

Здесь в методе `Строка.Содержит()`:

- Указан только первый аргумент — подстрока, которая ищется в исходной строке.
- Второй аргумент не указан, это значит, что сравнение строк будет выполняться с учётом регистра.

Узнать, содержатся ли в исходной строке только буквы, только цифры или только буквы и цифры

Это может понадобиться для проверки паролей, телефонов и подобных полей с обязательным заполнением.

Для этого используйте методы `Строка.ТолькоЦифры()`, `Строка.ТолькоБуквы()` и `Строка.ТолькоБуквыИлиЦифры()`:

```
метод Скрипт()
    пер Пароль = "Тест_123"
    если Пароль.ТолькоЦифры()
        // Сообщить, что пароль должен содержать не только цифры, но и буквы
    иначе если Пароль.ТолькоБуквы()
        // Сообщить, что пароль должен содержать не только буквы, но и цифры
    иначе если Пароль.ТолькоБуквыИлиЦифры()
        // Сообщить, что пароль должен содержать не только цифры и буквы, но и один из символов (_ ! -)
    иначе
        // Все правильно
    ;
;
```

Примечание:

Подробнее о типе `Строка` читайте в справке по стандартной библиотеке.

Типы для работы с датой и временем

Примечание:

Работа с датой и временем на базовом уровне описана здесь *(43)*.

Типы для работы с датой и временем

В «1С:Элементе» существует несколько типов для работы с датой и временем:

- `Время` — используйте тогда, когда дата не имеет значения, а важно только время. Например, для расписания занятий в течение дня.
- `Дата` — используйте тогда, когда время не имеет значения, а важна только дата. Например, для юбилеев, годовщин, дней рождения.
- `ДатаВремя` — используйте тогда, когда важны и дата и время. Например, в документах материального и финансового учёта (накладные, счета-фактуры).
- `Длительность` — используйте тогда, когда важно знать продолжительность интервала времени. Например, время в пути, продолжительность занятия и т. д.

Перечисленные типы удобны для представления информации пользователям. Однако они неудобны для фиксации событий, происходящих в разных временных зонах.

Событие, произошедшее в Москве в 13 часов дня и событие, произошедшее в Красноярске в этот же день в 13 часов дня, происходят не одновременно. Сначала происходит событие в Красноярске, а затем, через 4 часа, происходит событие в Москве.

Для учёта событий, происходящих в разных часовых поясах, существует тип `Момент` (абсолютное время). Он хранит дату и время, указанные для часового пояса UTC. Его использование позволяет правильно расположить события на оси времени:

- 13 часов дня, Красноярск: 2022-12-22T06:00:00.000Z
- 13 часов дня, Москва: 2022-12-22T10:00:00.000Z

Значения типа `Момент` неудобны для представления пользователям. Чтобы пользователь из Красноярска или пользователь из Москвы могли правильно ориентироваться во времени, абсолютное время (значение типа `Момент`) нужно преобразовать в локальное время пользователя (значение типа `ДатаВремя`). Для этого необходимо знать, в каком часовом поясе находится пользователь. Для указания часовых поясов существует ещё один тип — `ЧасовойПояс`.

Примечание:

Подробнее о типах для работы с датой и временем читайте:

- руководство разработчика,
- справку по стандартной библиотеке.

Написать в тексте скрипта значения даты и времени

Для записи значений даты и времени в тексте скрипта можно использовать следующие литералы:

- Тип `Время`

```
метод Скрипт()
    пер Время = Время{23:30:40}
;
```

В результате в переменной **Время** будет значение **23:30:40.000**.

- Тип `Дата`

```
метод Скрипт()
    пер Дата = Дата{2025-12-25}
;
```

В результате в переменной **Дата** будет значение **2025-12-25**.

- Тип `ДатаВремя`

```
метод Скрипт()
    пер ДатаИВремя = ДатаВремя{2025-12-25 23:30:40}
;
```

В результате в переменной **ДатаИВремя** будет значение **2025-12-25T23:30:40.000**;

- Тип `Длительность`

```
метод Скрипт()
    пер Длительность = 5д14ч30м
;
```

В результате в переменной **Длительность** будет значение **134:30:00.000**;

- Тип `Момент`

```
метод Скрипт()
    пер Момент = Момент{2025-12-22 13:00:00.000 UTC+7}
;
```

В результате в переменной **Момент** будет значение **2025-12-22T06:00:00.000Z**;

- Тип `ЧасовойПояс`

```
метод Скрипт()
    пер ЧасовойПояс = ЧасовойПояс{UTC+3}
;
```

В результате в переменной **ЧасовойПояс** будет значение **UTC+03:00**.

Значения даты и времени можно задать с помощью конструкторов: `новый Время()`, `новый Дата()` и т. д.

Если переменные типа `Дата`, `Время`, `ДатаВремя` и т. п. объявить без указания значения инициализации, то они примут следующие значения по умолчанию:

- Тип `Время`

```
метод Скрипт()
    пер ПустоеВремя: Время
;
```

  В результате в переменной **ПустоеВремя** будет значение **00:00:00.000**.

- Тип `Дата`

```
метод Скрипт()
    пер ПустаяДата: Дата
;
```

  В результате в переменной **ПустаяДата** будет значение **0001-01-01**.

- Тип `ДатаВремя`

```
метод Скрипт()
    пер ПустаяДатаИВремя: ДатаВремя
;
```

  В результате в переменной **ПустаяДатаИВремя** будет значение **0001-01-01T00:00:00.000**.

- Тип `Длительность`

```
метод Скрипт()
    пер ПустаяДлительность: Длительность
;
```

  В результате в переменной **ПустаяДлительность** будет значение **00:00:00.000**.

- Тип `Момент`

```
метод Скрипт()
    пер ПустойМомент: Момент
;
```

  В результате в переменной **ПустойМомент** будет значение **0001-01-01T00:00:00.000Z**.

У типа `ЧасовойПояс` нет значения по умолчанию.

Прибавить к дате нужное количество дней

Это может понадобиться, когда, зная начало отпуска и продолжительность, нужно получить конец отпуска.

Для этого используйте метод `Дата.ДобавитьДни()`:

```
метод Скрипт()
    пер НачалоОтпуска = Дата{2025-05-20}
    пер КонецОтпуска = НачалоОтпуска.ДобавитьДни(28)
;
```

В результате в переменной **КонецОтпуска** будет значение **2025-06-17**.

У других типов есть аналогичные методы: `Время.ДобавитьЧасы()`, `ДатаВремя.ДобавитьГоды()`, `ДатаВремя.ДобавитьСекунды()` и т. п.

Представить значения даты и времени в виде строки нужного формата

Это может понадобиться, чтобы показать дату и время в виде, удобном для пользователя.

Для этого используйте метод `ДатаВремя.Представление()`:

```
метод Скрипт()
    пер ДатаВремяСобытия = ДатаВремя{2025-01-12 12:20}
    пер Сообщение = "Событие запланировано на $ДатаВремяСобытия"
;
```

В результате в переменной **Сообщение** будет значение **"Событие запланировано на 12.01.2025 12:20:00"**;

Здесь метод `ДатаВремя.Представление()` неявно используется во время интерполяции *(167)* в строку сообщения значения переменной **ДатаВремяСобытия**. Дата и время по умолчанию показываются в формате «дд.ММ.гггг ЧЧ:мм:сс».

Чтобы задать своё представление даты и времени в соответствии с собственной форматной строкой, укажите её как аргумент метода `ДатаВремя.Представление()`:

```
метод Скрипт()
    пер ДатаВремяСобытия = ДатаВремя{2025-01-12 12:20}
    пер Сообщение = ДатаВремяСобытия.Представление("'Событие произойдёт 'дд ММММ гггг' в 'ЧЧ:мм' ('дддд')'")
;
```

В результате в переменной **Сообщение** будет значение **"Событие произойдёт 12 января 2025 в 12:20 (воскресенье)"**;

Здесь в методе `ДатаВремя.Представление()` указан единственный аргумент — форматная строка, в которой заданы:

- **дд** — день месяца (цифрами) с лидирующим нулём;
- **ММММ** — полное название месяца;
- **гггг** — полный номер года;
- **ЧЧ** — час в 24-часовом варианте с лидирующим нулём;
- **мм** — минута с лидирующим нулём;
- **дддд** — полное название дня недели;
- текст, указанный в одинарных апострофах, выводится как есть, без изменений.

Метод `ДатаВремя.Представление()` работает аналогичным образом для типов `Дата` и `Время`.

Получить день недели у даты (даты-времени)

Это может понадобиться, чтобы показать выходные дни особым образом.

Для этого используйте метод `Дата.ДеньНедели()`:

```
метод Скрипт()
    пер ДатаВстречи = Дата{2025-01-15}
    если ДатаВстречи.ДеньНедели() == ДеньНедели.Суббота или ДатаВстречи.ДеньНедели() == ДеньНедели.Воскресенье
        Консоль.Записать("Выходной день")
    иначе
        Консоль.Записать("Будний день")
    ;
;
```

В результате в терминал будет выведено сообщение **Будний день**.

Метод `Дата.ДеньНедели()` работает аналогичным образом и для типа `ДатаВремя`.

Кроме этого существуют методы `Дата.ДеньГода()` и `ДатаВремя.ДеньГода()`, которые возвращают порядковый номер дня в году.

Получить абсолютное время из локального

Это нужно для того, чтобы расположить на одной оси времени события, происходящие в разных часовых поясах.

Для этого используйте метод `ДатаВремя.ВМомент()`:

```
метод Скрипт()
    пер ЛокальноеВремяКрасноярск = ДатаВремя{2025-12-22 13:00:00.000}
    пер Момент = ЛокальноеВремяКрасноярск.ВМомент(ЧасовойПояс{UTC+7})
;
```

В результате в переменной **Момент** будет значение **2025-12-22T06:00:00.000Z**.

Здесь в методе `ДатаВремя.ВМомент()` указан единственный аргумент — часовой пояс Красноярска **UTC +7**, в котором задано локальное время. В результате метод возвращает абсолютное время в часовом поясе UTC.

Получить локальное время из абсолютного

Это нужно для того, чтобы вместо абсолютного времени использовать время, удобное пользователю.

Для этого используйте метод `Момент.ВДатаВремя()`:

```
метод Скрипт()
    пер Момент = Момент{2025-12-22 13:00:00.000 UTC+7}
    пер ЛокальноеВремя = Момент.ВДатаВремя(ЧасовойПояс.Текущий())
;
```

В результате в переменной **ЛокальноеВремя** будет значение **2025-12-22T09:00:00.000**;

Здесь в методе `Момент.ВДатаВремя()` указан единственный аргумент — текущий часовой пояс. Метод `ЧасовойПояс.Текущий()` является статическим, он вызывается просто через точку от имени типа.

Представить абсолютное время пользователю в его часовом поясе

Это нужно для того, чтобы абсолютное время представить в виде, удобном и понятном пользователю.

Для этого используйте метод `Момент.Представление()`:

```
пер Момент = Момент{2025-01-23 13:00:00.000 UTC+7}
пер ВремяДляПользователя = Момент.Представление()
```

В результате в переменной **ВремяДляПользователя** будет значение **"23.01.2025 09:00:00"**.

Здесь метод `Момент.Представление()` возвращает строковое представление момента в текущем часовом поясе в формате "дд.ММ.гггг ЧЧ:мм:сс". При этом неявно вызывается преобразование абсолютного момента времени в локальную `ДатаВремя` (**Момент.ВДатаВремя(ЧасовойПояс.Текущий())**), о котором рассказано в примере Получить локальное время из абсолютного *(185)*.

Другой способ — неявно использовать метод `Момент.Представление()` во время интерполяции значения переменной, содержащей момент, в строку. Для этого перед именем переменной указывается знак доллара «$».

```
пер Момент = Момент{2025-01-23 13:00:00.000 UTC+7}
пер Сообщение = "Событие произойдёт $Момент"
```

В результате в переменной **Сообщение** будет значение **"Событие произойдёт 23.01.2025 09:00:00"**.

Узнать продолжительность в днях

Это может понадобиться, когда нужно узнать продолжительность какого-то события в днях, часах, минутах и т. д.

Например, вам нужно узнать продолжительность отпуска в днях. Для этого найдите разность двух дат, содержащих начало и конец отпуска, и вызовите свойство `Длительность.Дни`:

```
метод Скрипт()
    пер НачалоОтпуска = Дата{2025-05-20}
    пер КонецОтпуска = Дата{2025-06-10}
    пер ПродолжительностьОтпуска = (КонецОтпуска - НачалоОтпуска).Дни
;
```

В результате в переменной **ПродолжительностьОтпуска** будет значение **21**.

Кроме этого вы можете получить значение типа `Длительность` как разность значений типа `Время`, `ДатаВремя`, `Момент` и вызвать свойства `Дни`, `Часы`, `Минуты`, `Секунды` и `Миллисекунды` у этой длительности.

Сравнить компоненты даты-времени

Это может понадобиться для того, чтобы узнать, в один ли день проходят события. Другой пример — узнать, в одно ли время проходят два разных события.

Для этого используйте свойства `ДатаВремя.Дата` и `ДатаВремя.Время`, чтобы сравнить их значения между собой:

```
пер ДатаВремя2 = ДатаВремя{2025-01-22 15:20:45}
пер ДатаВремя3 = ДатаВремя{2025-01-22 15:23:45}
если ДатаВремя2.Дата == ДатаВремя3.Дата и ДатаВремя2.Время != ДатаВремя3.Время
    Сообщить("События 2 и 3 проходят в один день, но в разное время")
;

пер ДатаВремя1 = ДатаВремя{2025-01-22 15:20:45}
пер ДатаВремя4 = ДатаВремя{2025-03-22 15:20:45}
если ДатаВремя1.Дата != ДатаВремя4.Дата и ДатаВремя1.Время == ДатаВремя4.Время
    Сообщить("События 1 и 4 проходят в одно и то же время, но в разные дни")
;
```

В результате в терминал будут выведены следующие сообщения:

```
События 2 и 3 проходят в один день, но в разное время
События 1 и 4 проходят в одно и то же время, но в разные дни
```

Помимо этого можно сравнивать отдельные компоненты даты и времени:

- с помощью свойств `Год`, `Месяц`, `День` у значений типа `Дата` и `ДатаВремя`;
- с помощью свойств `Час`, `Минута`, `Секунда`, `Миллисекунда` у значений типа `Время` и `ДатаВремя`.

Регулярные выражения

Общая информация

Существует множество задач, где могут использоваться регулярные выражения. Это и проверка строк на соответствие шаблонам (телефонные номера, электронная почта, номера договоров и т. п.), и сложный поиск в тексте документов, парсинг документов на соответствие требуемым шаблонам, и многое другое.

Механизм регулярных выражений в «1С:Элементе» состоит из трёх частей.

Во-первых, для описания регулярных выражений предназначен тип `Образец`. Экземпляр этого типа можно создать с помощью литерала или конструктора. Как это сделать, показано в примере Написать регулярное выражение в тексте скрипта *(188)*.

Во-вторых, у типа `Строка` существуют перегруженные *(212)* методы `Строка.Заменить()`, `Строка.Разделить()`, `Строка.Содержит()` и `Строка.ПолноеСовпадение()`, которые позволяют работать с регулярными выражениями. В качестве аргумента этих методов указывается не конкретная подстрока, а значение типа `Образец` (регулярное выражение), которому должна удовлетворять эта подстрока.

В-третьих, для хранения результатов поиска соответствий подстрок образцу используется экземпляр типа `Совпадение`. Массив значений типа `Совпадение` получается в результате выполнения метода `Образец.НайтиСовпадения()`.

> **Примечание:**
>
> Подробнее о работе с регулярными выражениями читайте в справке по стандартной библиотеке.

Написать регулярное выражение в тексте скрипта

Записать регулярное выражение в тексте скрипта можно с помощью литерала. Например:

```
пер РегулярноеВыражение = '(\ц{2}-){2}'
```

При использовании литерала строка, задающая конкретный образец, заключается в апострофы «'» и содержит специальные конструкции, приведённые в справке по стандартной библиотеке.

> **Примечание:**
>
> Апостроф «'» можно вводить с помощью Alt-ввода *(227)*.

Помимо этого регулярное выражение можно задать с помощью конструктора. Например:

```
пер РегулярноеВыражение = новый Образец("(\\ц{2}-){2}")
```

При использовании конструктора образец создаётся из строки, содержащей регулярное выражение и заключённой в двойные кавычки «"». Экранирование в регулярном выражении выполняется с помощью обратной косой черты «\». Экранировать необходимо только саму обратную косую черту «\» и одинарную кавычку «'».

Заменить в исходной строке вхождения подстроки по образцу

Это может понадобиться, когда заранее неизвестно, какую именно подстроку в исходной строке нужно заменить. Например, при показе пользователю номера банковской карты, в которой цифры разбиты по четыре и группы цифр отделены друг от друга дефисом «-», нужно скрыть (заменить на «****») все группы цифр, кроме последней.

Для этого используйте метод `Строка.Заменить()`:

```
метод Скрипт()
    пер НомерКарты = "1111-2222-3333-4444"
    НомерКарты = НомерКарты.Заменить('\ц{4}\Ц', "****-")
;
```

В результате в переменной **НомерКарты** будет значение **"****-****-****-4444"**.

Здесь в методе `Строка.Заменить()`:

- В первом аргументе указано регулярное выражение (тип `Образец`), которому должны удовлетворять подстроки, подлежащие замене. Образец **'\ц{4}\Ц'** означает четыре подряд идущих цифры и один любой нецифровой символ в качестве разделителя.
- Во втором аргументе указана подстрока замены.
- Третий аргумент не указан, это значит, что выполняются все замены в исходной строке.

В результате все группы цифр в номере карты, кроме последней, будут заменены звёздочками «****» и отделены друг от друга дефисом «-». Последняя группа цифр не будет заменена, так как после неё нет разделителя и она не соответствует образцу.

Разделить строку на части по образцу, содержащему перечень символов-разделителей

Это может понадобиться, когда заранее неизвестно, какой именно символ является разделителем в строке.

Например, есть фрагмент .csv-файла:

```
Имя;Фамилия;Телефон;Эл.Почта
Марина;Сотникова;89997775533;test@yandex.ru
Иван;Морозов;;test@mail.ru
Сергей;Краснов;89991112233;
```

Содержимое этого фрагмента находится в переменной **CSVСтрока**. Необходимо разделить этот текст на части по одному из символов-разделителей, перечисленных в образце.

Для этого используйте метод `Строка.Разделить()`:

```
метод Скрипт()
    пер CSVСтрока =
        "Имя;Фамилия;Телефон;Эл.Почта
        Марина;Сотникова;84956889002;test@yandex.ru
        Иван;Морозов;;test@mail.ru
        Сергей;Краснов;84956814407;"
    для ТекущаяСтрока из CSVСтрока.Разделить('[,;:\в\н]+')
        // Обработать текущую строку
        Консоль.Записать(ТекущаяСтрока)
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Имя
Фамилия
Телефон
Эл.Почта
Марина
Сотникова
84956889002
test@yandex.ru
Иван
Морозов
test@mail.ru
Сергей
```

```
Краснов
84956814407
```

Здесь в методе `Разделить()`:

- В первом аргументе указано регулярное выражение **'[,;:\в\н]+'** (тип `Образец`). Этот образец означает, что любой из символов, перечисленных в квадратных скобках (в том числе и символ новой строки — **\в\н**), может служить разделителем. Плюс «+» означает, что в тексте может идти несколько символов-разделителей подряд (чтобы игнорировать пустые строки).
- Второй аргумент не указан, это значит, что выполняются все возможные разбиения, при этом пустые строки, полученные в результате разбиения, идущие в конце массива, игнорируются.

Узнать, содержится ли подстрока по образцу в исходной строке

Это может понадобиться, когда имя файла содержит год его создания и нужно разным образом обработать файлы, созданные в разные годы.

Для этого используйте метод `Строка.Содержит()`:

```
метод Скрипт()
    пер ИмяФайла = "log_2025-10-31.txt"
    если ИмяФайла.Содержит('202\ц')
        // Скопировать файлы
    иначе если ИмяФайла.Содержит('201\ц')
        // Поместить файлы в архив
    иначе если ИмяФайла.Содержит('200\ц')
        // Удалить файлы
    ;
;
```

Здесь в методе `Строка.Содержит()` указан единственный аргумент — регулярное выражение (тип `Образец`), под который должна подходить подстрока, содержащаяся в имени файла. Символы «\ц» в регулярном выражении обозначают любую цифру от 0 до 9.

В результате:

- файлы, созданные в 2020-е годы (образец - **'202\ц'**), будут скопированы;
- файлы, созданные в 2010-е годы (образец - **'201\ц'**), будут помещены в архив;
- файлы, созданные в 2000-е годы (образец - **'200\ц'**), будут удалены.

Проверить, что строка полностью соответствует образцу

Это может понадобиться для проверки того, что введённая строка соответствует заданному шаблону. Например, нужно проверить, что номер телефона соответствует шаблону +7(XXX)XXX-XX-XX, где X — любая цифра от 0 до 9.

Для этого используйте метод `Строка.ПолноеСовпадение()`:

```
метод Скрипт()
    пер Телефон1 = "+7(495)688-90-02"
    пер Телефон2 = "8 495 681 44 07"
```

```
    пер ОбразецТелефона = '\+7\(\ц{3}\)\ц{3}(-\ц{2}){2}'
    пер Телефон1Совпадает = Телефон1.ПолноеСовпадение(ОбразецТелефона)
    пер Телефон2Совпадает = Телефон2.ПолноеСовпадение(ОбразецТелефона)
;
```

В результате:

- в переменной **Телефон1Совпадает** будет значение `Истина`;
- в переменной **Телефон2Совпадает** будет значение `Ложь`.

Если вы хотите, чтобы телефон соответствовал другому шаблону — 8 XXX XXX XX XX (где X — любая цифра от 0 до 9), то регулярное выражение будет выглядеть следующим образом:

```
метод Скрипт()
    пер Телефон1 = "+7(495)688-90-02"
    пер Телефон2 = "8 495 688 90 02"
    пер ОбразецТелефона = '8(\п\ц{3}){2}(\п\ц{2}){2}'
    пер Совпадает = Телефон1.ПолноеСовпадение(ОбразецТелефона) // Ложь
    Совпадает = Телефон2.ПолноеСовпадение(ОбразецТелефона) // Истина
;
```

В результате:

- в переменной **Телефон1Совпадает** будет значение `Ложь`;
- в переменной **Телефон2Совпадает** будет значение `Истина`.

Здесь в методе `ПолноеСовпадение()` указан единственный аргумент — регулярное выражение (тип `Образец`), под который должна подходить подстрока, содержащаяся в номере телефона.

Найти все совпадения в строке с заданным образцом

Это может понадобиться, чтобы найти в тексте все слова, которые начинаются на определённую последовательность символов.

Для этого задайте нужную последовательность в регулярном выражении и используйте метод `Образец.НайтиСовпадения()`:

```
метод Скрипт()
    пер Текст = "я ты он она мы вы они"
    пер РегулярноеВыражение = '\п(?=он)\с+'
    пер Совпадения = РегулярноеВыражение.НайтиСовпадения(Текст)
    для ОчередноеСовпадение из Совпадения
        Консоль.Записать(ОчередноеСовпадение.Значение())
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
он
она
они
```

Здесь:

- В переменной **РегулярноеВыражение** задан образец **'\п(?=он)\с+'**, который означает, что совпадением будут считаться все слова, начинающиеся на **он** и перед которыми стоит пробел.
- В методе `НайтиСовпадения()` первым аргументом указана исходная строка, в которой надо найти совпадения. Второй аргумент не указан, это значит, что будут найдены все совпадения в строке. Метод возвращает массив совпадений (тип `Массив<Совпадение>`).

Чтобы найти в строке **Текст** все слова, которые не начинаются с **он** и перед которыми стоит пробел, нужно указать образец **'\п(?!он)\с+'**.

```
метод Скрипт()
    пер Текст = "я ты он она мы вы они"
    пер РегулярноеВыражение = '\п(?!он)\с+'
    пер Совпадения = РегулярноеВыражение.НайтиСовпадения(Текст)
    для ОчередноеСовпадение из Совпадения
        Консоль.Записать(ОчередноеСовпадение.Значение())
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
ты
мы
вы
```

Удалить из строки все недопустимые символы, заданные в образце

Это может понадобиться, чтобы убрать все запрещённые символы из строки, содержащей имя файла.

Для этого задайте перечень недопустимых символов в регулярном выражении и используйте метод `Образец.НайтиСовпадения()`:

```
метод Скрипт()
    пер ИмяФайла = "123*Имя/Дата:777"
    пер РегулярноеВыражение = '[^\\/*:?]+'
    пер Совпадения = РегулярноеВыражение.НайтиСовпадения(ИмяФайла)
    пер НовоеИмяФайла: Строка
    для ОчередноеСовпадение из Совпадения
        НовоеИмяФайла += ОчередноеСовпадение.Значение()
    ;
;
```

В результате в переменной **НовоеИмяФайла** будет значение **123ИмяДата777**.

Здесь:

- В переменной **РегулярноеВыражение** задан образец **'[^\\/*:?]+'**, который означает, что совпадением будут считаться любые символы, кроме перечисленных в квадратных скобках после циркумфлекса «^». Обратная косая черта экранирована *(170)*.
- В методе `Образец.НайтиСовпадения()` первым аргументом указана исходная строка, в которой надо найти совпадения. Второй аргумент не указан, это значит, что будут найдены все совпадения в строке. Метод возвращает массив совпадений (тип `Массив<Совпадение>`).

В результате при обходе массива совпадений в переменной **НовоеИмяФайла** из исходной строки формируется строка, в которой убраны все недопустимые символы.

Поиск файлов по регулярному выражению

Регулярные выражения можно использовать не только при работе со строками, но и в некоторых других типах. Например, при поиске файлов на диске с помощью регулярного выражения можно задать образец, которому должно соответствовать имя файла.

Для этого используйте метод `НастройкиПоискаФайлов.ИмяСоответствует()`.

```
метод Скрипт()
    пер РегулярноеВыражение = '.*\.jpe?g'
    пер ПоискПоРегулярномуВыражению = новый НастройкиПоискаФайлов().ИмяСоответствует(РегулярноеВыражение)
    пер НайденныеФайлы = Файлы.Найти("D:\\test", ПоискПоРегулярномуВыражению)
;
```

В результате в массиве **НайденныеФайлы** будут файлы (тип `Файл`), у которых имена имеют расширения **JPG**, **jpg**, **jpeg** или **JPEG**.

> **Примечание:**
>
> Подробнее о работе с файлами читайте здесь *(245)*.

Работа с массивами

> **Примечание:**
>
> Работа с массивами на базовом уровне описана здесь *(110)*.

Удалить элемент

Удалить по значению

Чтобы удалить элемент массива по значению, используйте метод `Массив.Удалить()`.

```
метод Скрипт()
    пер Числа = [1, 5, 22, 5]
    Числа.Удалить(5)
;
```

В результате массив **Числа** будет содержать значение **[1, 22]**.

Здесь в методе `Массив.Удалить()`:

- Указан только первый аргумент — значение элемента, которое надо удалить из массива.
- Второй аргумент не указан, это значит, что будут удалены все вхождения элемента в массив.

Удалить по индексу

Чтобы удалить элемент массива по индексу, а не по значению, используйте метод `Массив.УдалитьПоИндексу()`:

```
метод Скрипт()
    пер Числа = [1, 5, 22, 7]
    Числа.УдалитьПоИндексу(1)
;
```

В результате массив **Числа** будет содержать значение **[1, 22, 7]**.

Удалить часть

Можно удалить из массива сразу несколько элементов в заданном диапазоне индексов. Для этого используйте метод Массив.УдалитьДиапазон():

```
метод Скрипт()
    пер Числа = [1, 5, 22, 7]
    Числа.УдалитьДиапазон(1, 3)
;
```

В результате массив **Числа** будет содержать значение **[1, 7]**. Удаляются элементы 5 и 22, т.к. верхний индекс не включается в диапазон.

Если во втором аргументе конечный индекс не задан, то удаляются все элементы до конца массива.

Удалить все

Чтобы удалить сразу все элементы массива, используйте метод `Массив.Очистить()`.

Добавить все элементы другого массива

Добавление в массив всех элементов другого массива может понадобиться при формировании маршрута.

Например, есть массив путевых точек — маршрут для проезда к текущему покупателю. Вы получили заказ от следующего покупателя, и теперь к имеющемуся маршруту нужно добавить маршрут к следующему покупателю — ещё один массив путевых точек.

Для этого используйте метод `Массив.ДобавитьВсе()`:

```
метод Скрипт()
    пер Маршрут = ["м. Красные ворота", "м. Лубянка", "м. Пушкинская"]
    пер Маршрут2 = ["м. Арбатская", "м. Киевская", "м. Парк Победы"]
    Маршрут.ДобавитьВсе(Маршрут2)
;
```

В результате массив **Маршрут** будет содержать значение **["м. Красные ворота", "м. Лубянка", "м. Пушкинская", "м. Арбатская", "м. Киевская", "м. Парк Победы"]**.

Здесь в методе `Массив.ДобавитьВсе()` указан единственный аргумент — коллекция, значения которой надо добавить в конец массива. Это может быть как массив, так и множество, но при этом тип элементов добавляемой коллекции должен совпадать с типом элементов исходного массива.

> **Примечание:**
>
> Для множества тоже доступен метод `Множество.ДобавитьВсе()`.

Вставить все элементы другого массива

Чтобы вставить в массив все элементы другой коллекции с заданного индекса, используйте метод `Массив.ВставитьВсе()`. Как это сделать, смотрите в примере Получить часть массива и вставить в другой массив *(197)*.

Удалить все элементы другого массива

Удаление всех элементов другого массива может понадобиться для обработки заявок.

Например, есть массив необработанных заявок. Они обходятся, какие-то заявки обрабатываются, какие-то нет. Создаётся массив тех заявок, которые уже обработаны. После этого в исходном массиве нужно оставить только необработанные заявки.

Для этого используйте метод `Массив.УдалитьВсе()`:

```
метод  Скрипт()
    пер  КодыЗаявок = [101, 103, 105, 106, 107]
    пер  ОбработанныеЗаявки = [107, 101, 105]
    КодыЗаявок.УдалитьВсе(ОбработанныеЗаявки)
;
```

В результате массив **КодыЗаявок** будет содержать значение **[103, 106]**.

Здесь в методе `Массив.УдалитьВсе()` указан единственный аргумент — коллекция, значения которой надо удалить из массива. Это может быть как массив, так и множество, но при этом тип элементов удаляемой коллекции должен совпадать с типом элементов исходного массива.

Чтобы, наоборот, оставить в массиве только элементы заданной коллекции, используйте метод `Массив.УдалитьКроме()`.

> **Примечание:**
>
> Для множества тоже доступны методы `Множество.УдалитьВсе()` и `Множество.УдалитьКроме()`.

Узнать, содержатся ли в массиве все элементы из другой коллекции

Это может понадобиться при подборе отелей, которые обладают некоторым набором услуг.

Например, у каждого отеля есть массив **ПредоставляемыеУслуги**, содержащий коды услуг, предоставляемых этим отелем. Есть массив **ВыбранныеУслуги**, содержащий коды услуг, которые нужны пользователю. Необходимо узнать, подходит ли отель под требования пользователя.

Для этого используйте метод `Массив.СодержитВсе()`:

```
метод Скрипт()
    пер ПредоставляемыеУслуги = [55, 56, 57, 60, 61]
    пер ВыбранныеУслуги = [60, 61, 57]
    если ПредоставляемыеУслуги.СодержитВсе(ВыбранныеУслуги)
        // Отель подходит
        Консоль.Записать("Отель подходит")
    иначе
        // Отель не подходит
        Консоль.Записать("Отель не подходит")
    ;
;
```

В результате в терминал будет выведено сообщение **Отель подходит**.

Здесь в методе `Массив.СодержитВсе()`:

- В первом аргументе указана коллекция, значения которой надо проверить на наличие в исходном массиве. Это может быть как массив, так и множество, но при этом тип элементов проверяемой коллекции должен совпадать с типом элементов исходного массива.
- Второй аргумент не указан, это значит, что дублирующиеся значения элементов проверяемой коллекции учитываться не будут.

> **Примечание:**
>
> Для множества тоже доступен метод `Множество.СодержитВсе()`.

Найти элемент

Это может понадобиться при синхронизации данных с внешней системой.

Например, в массиве **СписокЗадач** хранятся коды тех задач, которые уже есть в вашей системе. В переменной **ВнешняяЗадача** содержится код задачи, данные которой поступили из внешней системы. Нужно узнать, есть ли в вашей системе задача с таким кодом. Если есть — обновить её данные, если нет — добавить задачу.

Чтобы найти индекс первого вхождения элемента в массив, используйте метод `Массив.Найти()`:

```
метод Скрипт()
    пер СписокЗадач = [111, 77, 22, 203]
    пер ВнешняяЗадача = 22
    пер ИндексЗадачи = СписокЗадач.Найти(ВнешняяЗадача)
    если ИндексЗадачи != Неопределено
        // Обновить задачу
        Консоль.Записать("Обновить задачу")
    иначе
```

```
        // Добавить задачу
        Консоль.Записать("Добавить задачу")
    ;
;
```

В результате в терминал будет выведено сообщение **Обновить задачу**.

Здесь в методе `Массив.Найти()`:

- Указан только первый аргумент — элемент массива, который нужно найти.
- Второй и третий аргументы не указаны, это значит, что поиск будет выполняться во всём массиве.

Чтобы найти индекс последнего вхождения элемента в массив, используйте метод `Массив.НайтиСКонца()`.

Получить часть массива и вставить в другой массив

Это может понадобиться для обработки заказов.

Например, в массиве содержится список не обслуженных к концу дня заказов. Нужно разделить этот массив на части и вставить эти части в уже сформированные списки заказов курьеров на завтра.

Чтобы выделить часть массива, используйте метод `Массив.ПодМассив()`:

```
метод Скрипт()
    пер НеобслуженныеЗаказы = [145, 147, 135, 157, 151, 191, 190]
    пер ЗаказыДляКурьера2 = [24, 25]
    ЗаказыДляКурьера2.ВставитьВсе(0, НеобслуженныеЗаказы.ПодМассив(0, 4))
;
```

В результате в массиве **ЗаказыДляКурьера2** будет значение **[145, 147, 135, 157, 24, 25]**.

Здесь:

- В методе `Массив.ПодМассив()` в первом и во втором аргументах указаны начальный и конечный индексы, описывающие диапазон элементов исходного массива, которые будут скопированы в подмассив. Конечный индекс, **4**, не включается.
- В методе `Массив.ВставитьВсе()` в первом аргументе указан индекс элемента, начиная с которого вставляются элементы полученного подмассива в другой массив. Во втором аргументе указан вставляемый подмассив.

Сортировать элементы

Это может понадобиться для упорядочивания элементов массива по их значениям или по полю структуры, значения которой содержатся в массиве.

Простой вариант

Например, нужно отсортировать список фамилий сотрудников по алфавиту. Для этого используйте метод `Массив.СортироватьПо()`:

```
метод Скрипт()
    пер СписокСотрудников = ["Морозова", "Иванов", "Петров", "Сорокина", "Сидоров"]
```

```
    СписокСотрудников = СписокСотрудников.СортироватьПо(ФИО -> ФИО)
;
```

В результате в массиве **СписокСотрудников** будет значение **["Иванов", "Морозова", "Петров", "Сидоров", "Сорокина"].**

Здесь в методе `Массив.СортироватьПо()`:

- Указан только первый аргумент — лямбда-выражение *(218)*, которое возвращает значение элемента массива, по которому массив нужно отсортировать. В результате массив будет отсортирован по фамилиям сотрудников.
- Остальные аргументы не указаны. Это значит, что будут отсортированы все элементы массива в порядке возрастания.

Сложный вариант

Если элементы массива содержат, например, структуру, то с помощью лямбда-выражения можно указать поле этой структуры, по которому нужно отсортировать массив.

Например, массив **Студенты** состоит из элементов, каждый из которых описан в виде структуры **Студент**. Нужно отсортировать список студентов по полю **Оценка** этой структуры:

```
структура Студент
    знч ФИО: Строка
    знч Оценка: Число
;

метод Скрипт()
    пер Студенты = [новый Студент("Стрельцов",4),
                    новый Студент("Малахова", 5),
                    новый Студент("Соколов", 4),
                    новый Студент("Королев",3),
                    новый Студент("Мальцева", 5)]
    Студенты = Студенты.СортироватьПо(ПарамСтудент -> ПарамСтудент.Оценка, НаправлениеСортировки.ПоУбыванию)
;
```

В результате в массиве **Студенты** будет значение **[{"Малахова":5}, {"Мальцева":5}, {"Стрельцов":4}, {"Соколов":4}, {"Королев":3}]**.

Здесь в методе `Массив.СортироватьПо()`:

- В первом аргументе указано лямбда-выражение *(218)*, которое возвращает значение поля структуры (**Оценка**), по которому массив нужно отсортировать.
- Во втором аргументе указан порядок сортировки.

В результате для каждого экземпляра структуры **Студент**, содержащегося в массиве **Студенты**, извлекается значение поля структуры **Оценка**, и массив сортируется по этим значениям.

Работа с соответствиями

> **Примечание:**
>
> Работа с соответствиями на базовом уровне описана здесь *(130)*.

Удалить элемент

Чтобы удалить элемент соответствия по указанному ключу, используйте метод `Соответствие.Удалить()`:

```
метод Скрипт()
    пер ЭтапыРаботы = {"1 этап" : "Выполнено", "3 этап" : "Планируется", "2 этап" : "Выполняется"}
    ЭтапыРаботы.Удалить("3 этап")
;
```

В результате соответствие **ЭтапыРаботы** будет содержать значение **{"1 этап":"Выполнено", "2 этап":"Выполняется"}**.

Чтобы удалить сразу все элементы соответствия, используйте метод `Соответствие.Очистить()`.

Вставить в соответствие все элементы другого соответствия

Это может понадобиться для поддержания в актуальном состоянии данных какого-то соответствия.

Например, в соответствии содержатся данные о заказах товаров. Ключом соответствия является номер заказа, а значением — статус его выполнения. Нужно к концу дня актуализировать информацию о заказах: добавить новые и обновить существующие, не смотря на то, содержатся те или иные данные в соответствии или нет.

Для этого используйте метод `Соответствие.ВставитьВсе()`:

```
метод Скрипт()
    пер Заказы = {111:"Выполнен", 333:"В работе", 222:"Отменен", 777:"Выполнен", 555:"Перенесен"}
    Заказы.ВставитьВсе({99:"Создан", 333:"Выполнен", 66:"Создан", 555:"В работе"})
;
```

В результате соответствие **Заказы** будет содержать значение **{111:"Выполнен", 333:"Выполнен", 222:"Отменен", 777:"Выполнен", 555:"В работе", 99:"Создан", 66:"Создан"}**.

Здесь в методе `Соответствие.ВставитьВсе()` в виде литерала указан единственный аргумент — соответствие, значениями которого надо обновить исходное соответствие. При этом тип ключей и значений этого соответствия должен совпадать соответственно с типом ключей и значений исходного соответствия.

В результате в соответствие **Заказы** добавляются два новых заказа (с номерами **99** и **66**), а статусы заказов с номерами **333** и **555** обновляются.

Удалить некоторые элементы

Например, есть соответствие, хранящее номер заказа (ключ) и статус его выполнения (значение). Нужно удалить из соответствия все элементы, имеющие статус **"Выполнен"**.

Для перебора соответствия можно использовать цикл `для из`, но в этой инструкции нельзя модифицировать соответствие, нельзя удалять и добавлять элементы.

В такой ситуации используйте обход множества ключей соответствия, которое можно получить методом `Соответствие.Ключи()`.

```
метод Скрипт()
    пер Заказы = {111:"Выполнен", 333:"В работе", 222:"Отменен", 777:"Выполнен", 555:"Перенесен"}
    для Ключ из Заказы.Ключи()
        если Заказы[Ключ] == "Выполнен"
            Заказы.Удалить(Ключ)
        ;
    ;
;
```

В результате соответствие **Заказы** будет содержать значение **{333:"В работе", 222:"Отменен", 555:"Перенесен"}**.

Константы

Константа — это специальный вид переменной. Её нельзя изменить. Она объявляется в начале модуля и доступна во всех методах модуля.

```
конст ИМЯ_АРХИВА = "myZip.zip"
конст ОБЪЕМ = 22

метод Скрипт()
    пер ИмяФайла = ИМЯ_АРХИВА
    пер НовыйОбъем = УвеличитьОбъем()
;

метод УвеличитьОбъем(): Число
    возврат ОБЪЕМ + ОБЪЕМ * 0.2
;
```

В примере объявлено две константы: **ИМЯ_АРХИВА** и **ОБЪЕМ**. Первая используется в главном методе **Скрипт()**, вторая используется в методе **УвеличитьОбъем()**.

В «1С:Элементе» имена констант пишутся прописными буквами. Если имя состоит из нескольких слов, то между ними ставится нижнее подчёркивание «_».

При объявлении константы обязательно должно присутствовать значение инициализации.

Исключения

Общая информация

В «1С:Элементе» существует механизм обработки исключительных ситуаций. *Исключительная ситуация* — это ошибка, возникающая в процессе выполнения скрипта, или аналогичная проблема. Например, вы хотите прочитать файл, но на диске нет такого файла. Вы хотите получить элемент массива с индексом 7, но в массиве всего 3 элемента.

Исключение — это событие, которое возникает при появлении ошибки или другой внештатной ситуации. Говорят, что исключение *выбрасывается*.

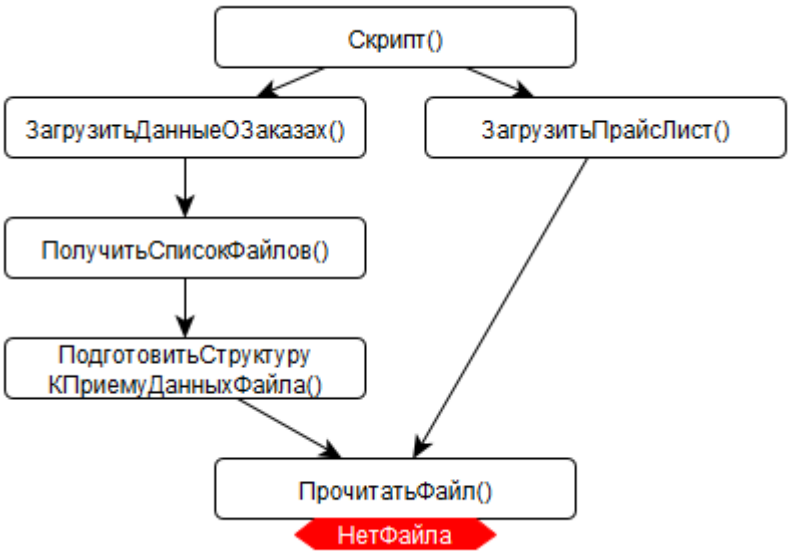
Выбросить исключение может «1С:Элемент» или вы. «1С:Элемент» содержит большое количество собственных исключений, которые он выбрасывает при возникновении тех или иных исключительных ситуаций. Точно так же вы можете выбросить любое из исключений «1С:Элемента» или собственное исключение. Вы можете выбросить исключение в какой-то момент работы вашего скрипта, если посчитаете, что произошла ошибка, препятствующая его дальнейшей работе.

В чём смысл выбрасывания исключений

Дело в том, что одна и та же ошибка, возникающая в разные моменты работы скрипта, может иметь разное значение и обрабатываться разным образом. Может даже быть так, что в одной ситуации эта ошибка не даёт возможность продолжить выполнение операции, а в другой ситуации можно её игнорировать и продолжать дальше.

Однако принять решение о том, как вести себя, в конкретной ситуации в месте возникновения ошибки бывает невозможно. Поэтому информацию о том, что выполнить заказанное действие не удалось, нужно передавать в программный код, который инициировал выполнение прервавшегося действия. Тогда уже вызывающий код будет принимать решение, что делать в случае возникновения той или иной ошибки.

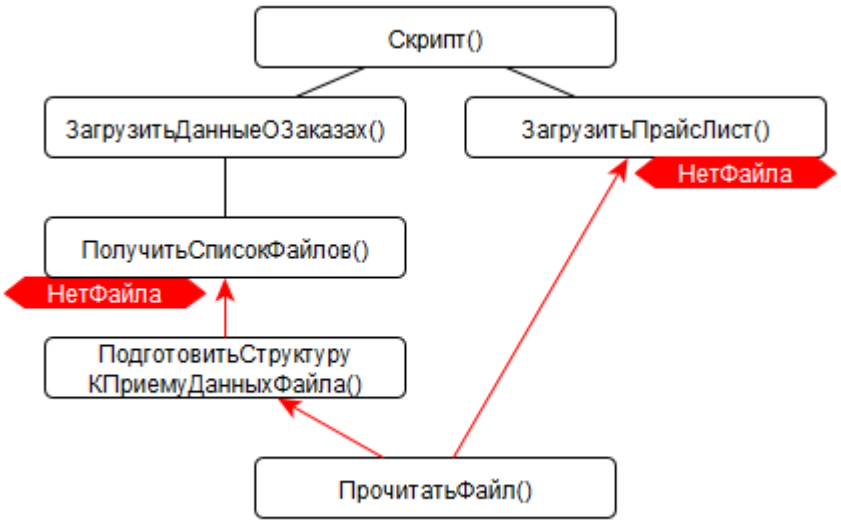
Вот наглядный пример. Скрипт выполняет много разных операций. В том числе он загружает данные о заказах — **ЗагрузитьДанныеОЗаказах()**, загружает прайс-лист — **ЗагрузитьПрайсЛист()**.



Данные о заказах — это набор отдельных файлов, каждый заказ в своём файле. Каждый файл нужно прочитать, а его данные поместить в структуру для дальнейшей обработки.

Прайс-лист — это многотомный архив, состоящий из нескольких файлов. Чтобы получить то, что находится в архиве, нужно прочитать все файлы, объединить их вместе и разархивировать.

И одна, и другая операция в какой-то момент времени приходят в один и тот же метод, который читает файл с диска. Вдруг оказывается, что такого файла на диске нет. Что делать дальше?



Находясь в методе **ПрочитатьФайл()**, вы этого не знаете, т.к. непонятно, каким из двух путей вы попали в этот метод.

Механизм исключений в «1С:Элементе» устроен таким образом, что, когда выбрасывается исключение, работа скрипта на текущем уровне вложенности прерывается. Управление передаётся на предыдущий уровень (в предыдущий метод) по порядку вызова методов «1С:Элемента» (по стеку вызовов).

Это очень хорошо. Получается, что если вы читали архив прайс-листа, то исключение вернётся в метод **ЗагрузитьПрайсЛист()**. Находясь в этом методе, вы знаете, что если какой-то файл не найден, то дальше продолжать нет смысла, архив распаковать не удастся. Вы завершаете работу.

Если же вы загружали данные о заказах, то это исключение поднимется в метод **ПолучитьСписокФайлов()**. В этом методе вы получили список файлов и обходите его по одному. Тут вы знаете, что если очередной файл не найден, то это не страшно, можно продолжать читать другие файлы, загружать данные из них. Вы продолжаете работу.

Пример использования исключения

```
исключение ИсключениеНеверныйФорматФайла
    пер ИмяФайла: Строка
;

метод Скрипт()
    попытка
        МойМетод()

    поймать Искл: ИсключениеНеверныйФорматФайла
        Консоль.Записать(Искл.Описание + " " + Искл.ИмяФайла + " " + Искл.ПоследовательностьВызовов)
    ;
;

метод МойМетод()
    выбросить новый ИсключениеНеверныйФорматФайла("Неверный формат файла", "D:\\test.xml")
;
```

Чтобы использовать не одно из стандартных исключений «1С:Элемента», а собственное, его нужно объявить. Объявление исключения похоже на объявление структуры (подробнее (204)).

```
исключение ИсключениеНеверныйФорматФайла
    пер ИмяФайла: Строка
;
```

Код скрипта, который может привести к выбрасыванию этого исключения, вы помещаете внутрь инструкции `попытка` (подробнее (204)). Она имеет секции, как инструкция `если`. В секцию `попытка` вы помещаете код, который может вызвать выброс исключения. Не важно, на каком уровне вложенности вызовов это произойдёт.

В секцию `поймать` вы помещаете код, который должен отработать в случае выброса исключения. То есть тот код, который должен «решить» возникшую проблему. Для этого вы указываете тип исключения, которое будете обрабатывать в этой секции. Таких секций может быть несколько — например, каждая для своего типа исключения.

```
метод Скрипт()
    попытка
        МойМетод()

    поймать Искл: ИсключениеНеверныйФорматФайла
        Консоль.Записать(Искл.Описание + " " + Искл.ИмяФайла + " " + Искл.ПоследовательностьВызовов)
    ;
;
```

В том месте скрипта, где возникает ошибка или происходит аналогичная проблема, вы выбрасываете исключение. Для этого вы пишете инструкцию `выбросить` (подробнее (206)).

```
метод МойМетод()
    выбросить новый ИсключениеНеверныйФорматФайла("Неверный формат файла", "D:\\test.xml")
;
```

В этой инструкции вы создаёте экземпляр своего исключения и выбрасываете его.

«Плохая » последовательность исполнения кода будет следующей. Сначала исполнение заходит в секцию `попытка` и проваливается в метод **МойМетод()**.

Тут выбрасывается исключение, исполнение кода прерывается и возвращается в метод **Скрипт()**.

Здесь оно проваливается в секцию **поймать Искл: ИсключениеНеверныйФорматФайла** и выполняются инструкции, размещённые в этой секции.

После этого исполнение переходит на инструкцию, следующую за инструкцией `попытка`.

«Хорошая» последовательность исполнения кода отличается тем, что весь код, расположенный в секции `попытка`, выполняется полностью. После этого исполнение переходит на инструкцию, следующую за инструкцией `попытка`.

Объявление исключения

Исключение объявляется в начале модуля, до методов. Объявление начинается с ключевого слова `исключение` и заканчивается точкой с запятой «;». После ключевого слова идёт имя типа исключения и описываются его поля.

Поля описываются аналогично тому, как объявляются переменные *(23)*: модификатор, имя и тип. Дополнительно можно указать ключевое слово `обз`. Например:

```
исключение ИсключениеЧтенияHTTPФайла
    обз знч ПутьКФайлу: Строка
    знч КодОтвета: Число
    пер ВремяОтветаСервера: Длительность
;
```

В этом примере объявляется тип исключения с именем **ИсключениеЧтенияHTTPФайла**, у которого есть три поля: **ПутьКФайлу**, **КодОтвета** и **ВремяОтветаСервера**.

Поля **ПутьКФайлу** и **КодОтвета** изменять нельзя (модификатор `знч`).

«1С:Элемент» автоматически создаёт для вашего исключения конструктор, который включает в себя все поля исключения. Значения полей, которые вы отметили как `обз`, **ПутьКФайлу**, обязательно должны быть указаны в конструкторе, значения остальных полей — по желанию.

> **Примечание:**
>
> Вы можете в обязательном порядке использовать в конструкторе только именованные аргументы. Как это сделать, смотрите здесь *(210)*.

Инструкция «попытка»

Инструкция `попытка` позволяет выполнить код и перехватить и обработать все исключения, которые могут возникнуть в процессе его выполнения.

Для перехвата исключений предназначены секции этой инструкции — `поймать`. Каждая секция предназначена для перехвата одного или нескольких типов исключений. В любом случае в объявлении этой секции указывается переменная, в которую будет помещён экземпляр выброшенного исключения.

Секция, перехватывающая один тип исключения **ИсключениеНеверныйФорматФайла** в переменную **Искл1**, может выглядеть так.

```
поймать Искл1: ИсключениеНеверныйФорматФайла
    ОбработатьНеверныйФорматФайла(Искл1)
```

Секция, перехватывающая два типа исключений **ТипИсключения1** или **ТипИсключения2** в переменную **Искл2**, может выглядеть так.

```
поймать Искл2: ТипИсключения1| ТипИсключения2
    ОбработатьОдноИлиДругоеИсключение(Искл2)
```

Все типы исключений: и встроенные исключения «1С:Элемента», и исключения, объявленные вами — находятся в общей иерархии типов. Базовым типом для них всех является тип `Исключение`. Поэтому, чтобы перехватывать вообще все исключения, которые могут возникнуть, в последней секции инструкции попытка вы можете написать перехват исключения типа `Исключение`.

```
поймать Искл3: Исключение
    // Обработать все другие исключения.
```

Помимо секций `поймать` в инструкции `попытка` есть завершающая секция `вконце`. В ней можно поместить код, который должен выполнить какие-то действия в любом случае, вне зависимости от того, удачно или неудачно завершился блок `попытка`.

Блок `вконце` выполняется даже в том случае, если в блоках `попытка` и `поймать` используются инструкции `возврат`, `прервать` или `продолжить`.

```
вконце
    // Завершающая секция.
```

Пример инструкции `попытка`, перехватывающий несколько специальных исключений и все остальные исключения, может выглядеть следующим образом.

```
исключение ИсключениеНеверныйФорматФайла
    пер ИмяФайла: Строка
;

исключение Исключение1
    пер ИмяФайла: Строка
;

исключение Исключение2
    пер ИмяФайла: Строка
;

метод МойМетод()
    попытка
        // Безопасный код.
        пер Файл = новый Файл("C:\\test\\test.txt")
        исп Поток = Файл.ОткрытьПотокЧтения()
        пер ТекстФайла = Поток.ПрочитатьКакСтроку()

    поймать Искл1: ИсключениеНеверныйФорматФайла
        // Обработать исключение типа ИсключениеНеверныйФорматФайла
        ОбработатьНеверныйФорматФайла(Искл1)

    поймать Искл2: ТипИсключения1| ТипИсключения2
        // Обработать исключения с типом Исключение1 или Исключение2

    поймать Искл3: Исключение
        // Обработать все другие исключения.

    вконце
        // Завершающая секция.
    ;
;
```

```
метод ОбработатьНеверныйФорматФайла(Искл: ИсключениеНеверныйФорматФайла)
;
```

Инструкция «выбросить»

Инструкция `выбросить` имеет единственный аргумент — экземпляр, описывающий выбрасываемое исключение. Вы создаёте такой экземпляр с помощью конструктора.

Для «ваших» типов исключений, которые вы объявили в скрипте, вы всегда можете это сделать. Например:

```
исключение ИсключениеЧтенияФайла
    пер ИмяФайла: Строка
;

метод Скрипт()
    попытка
        пер Файл = новый Файл("C:\\test\\test.txt")
        если не Файл.Существует()
            выбросить новый ИсключениеЧтенияФайла("Файл не существует", "C:\\test\\test.txt")
        ;
        исп Поток = Файл.ОткрытьПотокЧтения()
        пер ТекстФайла = Поток.ПрочитатьКакСтроку()
    поймать Искл: ИсключениеЧтенияФайла
        Консоль.Записать(Искл.Описание + " " + Искл.ИмяФайла)
    ;
;
```

В результате в терминал будет выведено сообщение **Файл не существует C:\test\test.txt**.

Для встроенных исключений «1С:Элемента» не всегда можно вызвать конструктор, поэтому вы сможете выбросить только те исключения, для которых в справке по стандартной библиотеке описан конструктор. Например, **ИсключениеИндексВнеГраниц**.

```
метод Скрипт()
    пер МойМассив = <Число>[]
    попытка
        // Установить значение элемента массива.
        если МойМассив.Граница() == -1
            выбросить новый ИсключениеИндексВнеГраниц("В массиве нет ни одного элемента")
        иначе
            МойМассив[0] = 5
        ;
    поймать Искл: ИсключениеИндексВнеГраниц
        Консоль.Записать(Искл.Описание)
    ;
;
```

В результате в терминал будет выведено сообщение **В массиве нет ни одного элемента**.

Если бы вы просто попытались изменить значение элемента пустого массива, вы получили бы сообщение **Индекс 0 находится за пределами пустой коллекции**.

Список типов встроенных исключений, для которых существуют конструкторы, смотрите в руководстве разработчика.

> **Примечание:**
>
> Подробнее об исключениях:
>
> - руководство разработчика,
> - справка по стандартной библиотеке.

Работа с методами

> **Примечание:**
>
> Работа с методами на базовом уровне описана здесь *(81)*.

Значения параметров по умолчанию

Как вы помните, при вызове метода ему можно передать набор аргументов — значений, которые этот метод должен обработать. Внутри метода эти значения доступны в виде параметров, перечисленных в описании метода.

Обычно передаваемые аргументы соответствуют перечисленным параметрам. То есть первый аргумент передаётся в первый параметр, второй — во второй, и так далее.

Порой есть аргументы, которые меняются очень редко. Например, метод архивирует файл. Он получает путь, где должен находиться архив, путь, где находится файл, который нужно добавить в архив, и степень сжатия, которую нужно использовать для добавления файла в архив.

```
метод Скрипт()
    ЗаписатьВАрхив("D:\\archives\\test.zip", "D:\\test\\test.xml", УровеньСжатияZip.Нормальный)
;

метод ЗаписатьВАрхив(АрхивПуть: Строка, ФайлПуть: Строка, Сжатие: УровеньСжатияZip)
    пер АрхивФайл = новый ФайлZip(АрхивПуть)
    АрхивФайл.Добавить(новый Файл(ФайлПуть), "", Сжатие)
;
```

Особенность этого метода заключается в том, что чем выше степень сжатия, тем меньше размер архива, но тем больше время, которое нужно для того, чтобы сжать и добавить файл. Обычно всегда используется степень сжатия `УровеньСжатияZip.Нормальный`, потому что она оптимальна и по времени, и по размеру архива. То есть получается не очень большой архив за приемлемое время.

Совсем отказаться от управления степенью сжатия в этом методе вы не можете, потому что бывают ситуации, когда нужно сжать файл как можно сильнее. Ради этого вы готовы ждать любое количество времени. Поэтому в методе есть третий параметр, но меняется он очень редко.

В такой ситуации вам поможет указание значений параметров по умолчанию.

Вы можете не писать каждый раз в вызове метода аргумент, который обычно не меняется, — **УровеньСжатияZip.Нормальный**.

```
ЗаписатьВАрхив("D:\\archives\\archive.zip", "D:\\test\\test.xml", УровеньСжатияZip.Нормальный)
```

Вместо этого можно один раз задать значение по умолчанию этого параметра в объявлении метода — **Сжатие = УровеньСжатияZip.Нормальный**.

```
метод ЗаписатьВАрхив(АрхивПуть: Строка, ФайлПуть: Строка, Сжатие = УровеньСжатияZip.Нормальный)
```

Тогда, если в вызове метода этот аргумент пропущен, параметр будет иметь значение по умолчанию. В редких случаях, когда требуется максимальное сжатие, вы будете указывать его в явном виде.

```
метод Скрипт()
    ЗаписатьВАрхив("D:\\archives\\test.zip", "D:\\test\\test.xml")    // обычное архивирование
    ЗаписатьВАрхив("D:\\archives\\big.zip", "D:\\test\\big.xml", УровеньСжатияZip.Максимальный)    // максимал
;

метод ЗаписатьВАрхив(АрхивПуть: Строка, ФайлПуть: Строка, Сжатие = УровеньСжатияZip.Нормальный)
    пер АрхивФайл = новый ФайлZip(АрхивПуть)
    АрхивФайл.Добавить(новый Файл(ФайлПуть), "", Сжатие)
;
```

Аргументы, которым соответствует параметр с указанным значением по умолчанию, называются *необязательными аргументами*.

Аргументы, которым соответствует параметр без значения по умолчанию, называются *обязательными аргументами*.

Когда вы устанавливаете значения параметров по умолчанию, нужно помнить о следующей особенности. Установить эти значения можно для любых параметров, но имеет смысл это делать только для тех параметров, которые расположены в конце списка.

При вызове метода можно не указывать только те необязательные аргументы, которые расположены в конце списка после последнего обязательного аргумента. Все остальные аргументы нужно указывать в явном виде, даже если они необязательные.

Например, в следующем примере указание второго аргумента **60** необходимо, несмотря на то что для его параметра **Глубина = 60** указано значение по умолчанию.

```
метод Скрипт()
    ДобавитьТовар(180, 60, 30)
;

метод ДобавитьТовар(Длина: Число, Глубина = 60, Ширина: Число, Цвет = "белый")
;
```

Из этого правила есть исключения. Вы узнаете о них в разделе Позиционные и именованные аргументы *(209)*.

Позиционные и именованные аргументы

До сих пор вы имели дело только с позиционными аргументами. *Позиционный аргумент* — это аргумент, который предаётся в параметр, находящийся в той же позиции. Первый аргумент передаётся в первый параметр, второй — во второй, и так далее.

Именованные аргументы

Кроме этого существует другой способ — передача аргументов по имени. Прямо в вызове метода вы можете указать имя параметра, в который нужно передать аргумент. Тогда становится неважно, в какой последовательности перечислены аргументы.

```
метод Скрипт()
    ДобавитьТовар(180, 30, 60)                                  // позиционные аргументы
    ДобавитьТовар(Глубина = 60, Длина = 180, Ширина = 30)   // именованные аргументы
;

метод ДобавитьТовар(Длина: Число, Ширина: Число, Глубина: Число)
;
```

Именованный аргумент — это аргумент, для которого указано имя параметра, в который его нужно передать.

Форма записи именованного аргумента состоит из имени параметра, знака равенства «=» и самого аргумента. Например, **Ширина = 30**.

Именованные и позиционные аргументы вместе

Позиционные и именованные аргументы можно использовать одновременно. В таком случае все именованные аргументы должны идти после позиционных.

Например, в следующем примере первый аргумент, **180**, — позиционный, а остальные — именованные:

```
метод Скрипт()
    ДобавитьТовар(180, Глубина = 60, Ширина = 30)
;

метод ДобавитьТовар(Длина: Число, Ширина: Число, Глубина: Число)
;
```

Пропуск необязательных аргументов

Как вы узнали из предыдущего раздела *(207)*, пропускать необязательные позиционные аргументы можно тогда, когда они идут в конце списка.

Если вы используете только именованные аргументы, то можно пропускать любые необязательные аргументы. Например, можно пропустить аргумент **Ширина**.

```
метод Скрипт()
    ДобавитьТовар(Глубина = 60, Длина = 180)
;
```

```
метод ДобавитьТовар(Длина: Число, Ширина: Число = 30, Глубина: Число)
;
```

Если вы используете одновременно и позиционные, и именованные аргументы, то за пропущенными аргументами должны следовать только именованные. Например, первый аргумент, 180, позиционный, второй опущен, а третий, **Глубина**, именованный.

```
метод Скрипт()
    ДобавитьТовар(180, Глубина = 60)
;

метод ДобавитьТовар(Длина: Число, Ширина: Число = 30, Глубина: Число)
;
```

Именованные аргументы в системных методах

Передачу аргументов по имени можно использовать в любых системных методах «1С:Элемента». Например, если вам так удобнее, можно передавать в метод `Строка.Заменить()` сначала добавляемый фрагмент, а потом тот фрагмент, который нужно заменить.

```
метод Скрипт()
    пер Источник = "позиционный параметр"
    Источник = Источник.Заменить(Замена = "именованный", Строка = "позиционный")
;
```

Принудительное использование именованных аргументов

Вы можете указать в скрипте, что в конкретном методе или конструкторе можно использовать только именованные аргументы. Для этого используйте аннотацию `@ИменованныеПараметры`.

Что такое аннотации, вы подробнее узнаете в разделе Модульная разработка *(213)*. Сейчас достаточно представить, что это специальная синтаксическая конструкция, которую можно написать, например, перед объявлением собственного метода.

```
метод Скрипт()
    ДобавитьТовар(Глубина = 60, Длина = 180, Ширина = 30)
;

@ИменованныеПараметры
метод ДобавитьТовар(Длина: Число, Ширина: Число, Глубина: Число)
;
```

Эту аннотацию можно использовать в описаниях структур и исключений. Для этого в описание структуры или исключения нужно добавить ключевое слово `конструктор`, перед которым написать аннотацию `@ИменованныеПараметры`.

Например, в своей структуре **Товар** вы можете указать, что при её конструировании нужно обязательно использовать именованные аргументы.

```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
```

```
    пер Цена: Число = 1

    @ИменованныеПараметры
    конструктор
;

метод Скрипт()
    пер ПервыйТовар = новый Товар(Цена = 5300, Наименование = "Холодильник")
;
```

Передача аргументов по значению

В «1С:Элементе» в переменных хранятся ссылки на экземпляры. Когда вы передаёте переменную в метод, передаётся именно значение этой переменной, экземпляр.

Это значит, что, находясь в методе, вы не можете изменить значение переменной, передаваемой в качестве аргумента.

Например, вы передаёте в метод переменную **Длина**, которая имеет значение **80**. В методе вы увеличиваете переданное значение. После возврата из метода переменная **Длина** по-прежнему имеет значение **80**:

```
метод Скрипт()
    пер Длина = 80
    Удлиннить(Длина)
    Консоль.Записать("Длина = " + Длина)
;

метод Удлиннить(Длина: Число)
    Длина += 100
;
```

В результате в терминал будет выведено сообщение **Длина = 80**.

В то же время вы можете изменять состояние экземпляров, переданных через параметры, с помощью методов этих экземпляров.

Например, в переданный массив **МассивНомеров** можно добавить новый элемент, и это новое значение вы увидите после возврата из метода:

```
метод Скрипт()
    пер МассивНомеров = ["первый", "второй", "третий"]
    Удлиннить(МассивНомеров)
    Консоль.Записать("Номера = " + МассивНомеров.Представление())
;

метод Удлиннить(МассивНомеров: Массив<Строка>)
    МассивНомеров.Добавить("пятый")
;
```

В результате в терминал будет выведено сообщение **Номера = [первый, второй, третий, пятый]**.

Перегрузка метода

Перегрузка метода — это использование методов с одинаковыми именами, но с разными параметрами, в одной области видимости имён *(94)*. Отличаться может количество параметров, отличаться могут типы тех же самых параметров или отличаться может и то и другое.

Тип возвращаемого значения не влияет на возможность перегрузки. То есть не может быть двух методов, которые отличаются только типом возвращаемого значения.

Например, перегруженный метод **МойМетод()** может возвращать значение того же типа `Строка`, но содержать два параметра вместо одного:

```
метод Скрипт()
    пер Результат  = ВернутьФразу("Ок")
    пер Перегрузка = ВернутьФразу(7, "Ок")
;

метод ВернутьФразу(Образец: Строка): Строка
    возврат Образец
;

метод ВернутьФразу(Количество: Число, Образец: Строка): Строка
    пер Результат = ""
    для Счетчик = 1 по Количество
        Результат = Результат + Образец + ", "
    ;
    возврат Результат
;
```

В результате в переменной **Результат** будет значение **"Ок"**, а в переменной **Перегрузка** будет значение **"Ок, Ок, Ок, Ок, Ок, Ок, Ок, "**.

Если бы возможность перегрузки отсутствовала, вам пришлось бы делать два разных метода с разными именами: **ВернутьФразу()** и **ВернутьФразуСПовторениями()**. Либо вам пришлось бы делать один метод с двумя параметрами, а внутри анализировать количество переданных параметров. И в зависимости от этого реализовывать либо один, либо другой алгоритм.

С использованием перегрузки скрипт выглядит лаконичнее и понятнее.

Другой пример перегрузки методов — когда перегруженный метод отличается только типом параметра. Вместо `Строка` в метод **СообщитьДату()** можно передать значение типа `Дата`.

```
метод Скрипт()
    пер Результат  = СообщитьДату("25.06.2025")
    пер Перегрузка = СообщитьДату(ДатаВремя.Сейчас().Дата)
;

метод СообщитьДату(Дата: Строка): Строка
    возврат "Сегодня " + Дата
;

метод СообщитьДату(Дата: Дата): Строка
```

```
    возврат Дата.Представление("'Сегодня 'дд ММММ гггг' ('дддд')'")
;
```

В результате в переменной **Результат** будет значение **"Сегодня 25.06.2025"**, а в переменной **Перегрузка** будет значение **"Сегодня 25 июня 2025 (среда)"**.

Есть ряд ситуаций, когда «1С:Элемент» не сможет определить используемый вариант метода. В этом случае произойдёт ошибка компиляции:

- Если у перегруженного метода тип одного параметра совпадает с типом параметра исходного метода, а второй параметр имеет значение по умолчанию.
- Если у перегруженного метода тип параметра является наследником типа параметра исходного метода.
- Если у перегруженного метода параметр описан с использованием составного типа, при этом в каждом варианте метода в типе параметра присутствует один и тот же тип.
- Если у перегруженного метода отличается только тип возвращаемого значения.

Подробнее об этих случаях читайте в руководстве разработчика.

Модульная разработка

Общая информация

«1С:Элемент» поддерживает модульный подход к разработке и исполнению скриптов. Это значит, что весь код, который вы хотели бы исполнять, может находиться не в одном, а в нескольких файлах. Каждый из этих файлов выполняет определённую функцию и имеет небольшой размер.

Такой подход позволяет:

- **Разделить огромные скрипты на более мелкие**. Просмотр и отладка одного большого файла сложнее, чем работа с несколькими небольшими файлами.
- **Повторно использовать код в других скриптах**. Разные скрипты могут использовать одну и ту же функциональность. Чтобы не дублировать её в каждом скрипте, можно вынести эти методы в отдельный файл и подключать этот файл к тем скриптам, где эта функциональность необходима.
- **Обеспечить изоляцию кода**. Часть методов, расположенных в файле, можно скрыть от других скриптов, предоставив другим скриптам только некоторые методы.
- **Упростить отладку и тестирование скриптов**. Для небольших алгоритмов проще подбирать параметры тестирования и проще расследовать проблемы.
- **Повысить скорость разработки за счёт параллельной работы над отдельными скриптами**. Если в разработке принимают участие несколько человек, то каждый может работать над своим или своими файлами, которые потом подключаются к главному скрипту.

Аннотации

Для того чтобы описать видимость методов для других скриптов, в «1С:Элементе» используются аннотации. *Аннотация* — это синтаксическая конструкция, которая позволяет добавить в программный код дополнительную информацию о самом этом программном коде. Такая информация предназначена

компилятору «1С:Элемента» и другим его механизмам. Эта информация сообщает им, что данную конструкцию «1С:Элемента» нужно обработать особенным образом, не так, как остальные.

Аннотации пишутся с новой строки, начинаются со знака "коммерческое а" «@». За ним следует имя аннотации. Для указания области видимости используются аннотации `@Глобально` и `@Локально`. Например:

```
@Глобально
метод  МойМетод()
      ВыводСообщения()
;
```

> **Примечание:**
>
> Знак "коммерческое а" «@» можно вводить с помощью Alt-ввода *(227)*.

Подключаемый скрипт

Чтобы иметь возможность пользоваться методами подключаемого скрипта, нужно описать их видимость с помощью аннотаций.

Аннотация `@Глобально` означает, что следующий за ней метод будет виден для других скриптов. Аннотация `@Локально` означает, что следующий за ней метод будет виден только в том скрипте, где он описан.

Таким образом, с помощью аннотаций области видимости вы можете создать «внешний» программный интерфейс вашего скрипта и его «внутреннюю» реализацию. Например:

```
@Глобально
метод  МойМетод()
      ВыводСообщения()
;

@Локально
метод  ВыводСообщения()
      Консоль.Записать("Мой  метод")
;
```

В этом примере внешний интерфейс представлен единственным методом **МойМетод()**. К этому методу можно будет обращаться из других скриптов. Внутренняя реализация представлена тоже единственным методом **ВыводСообщения()**. Этот метод не виден другим скриптам и недоступен им.

При создании подключаемых скриптов нужно учитывать, что циклические зависимости между скриптами запрещены. Другими словами, вы не можете создать скрипт «А», который требует подключения скрипта «В», в свою очередь зависящего от скрипта «А».

Подключение к главному скрипту

Чтобы к главному скрипту подключить другой скрипт, используйте *директиву импорта* `#требуется`. Эта директива пишется в начале скрипта, до методов. Например:

```
#требуется  модуль.sbsl

метод  Скрипт()
    модуль.МойМетод()
;
```

В этом примере к скрипту подключается другой скрипт с именем **модуль.sbsl**, который расположен в том же каталоге. В данном случае задан относительный путь к файлу скрипта, но можно задавать и абсолютный путь.

В результате такого описания после подключения скрипта **модуль.sbsl** «1С:Элемент» создаст для него тип-одиночку *(227)* с таким же именем, как имя файла, — `модуль`. Этот тип позволяет получить доступ к публичному интерфейсу подключаемого скрипта: к методам, к пользовательским типам (структурам, исключениям и пр.). Для этого нужно указать имя метода или типа через точку от этого типа.

Следующий пример показывает, как вызвать метод **МойМетод()**, определённый в скрипте **модуль.sbsl**. Файлы **скрипт.sbsl** и **модуль.sbsl** находятся в одном каталоге.

скрипт.sbsl

```
#требуется  модуль.sbsl

метод  Скрипт()
    модуль.МойМетод()
;
```

модуль.sbsl

```
@Глобально
метод  МойМетод()
    ВыводСообщения()
;

@Локально
метод  ВыводСообщения()
    Консоль.Записать("Мой  метод")
;
```

Методы с аннотацией `@Глобально` создают интерфейс скрипта, которым можно пользоваться «снаружи» — **модуль.МойМетод()**.

Методы с аннотацией `@Локально` используются для реализации внутренней функциональности, не видны из других модулей и недоступны из них.

> **Примечание:**
>
> Скрипт также можно подключить с помощью переменной окружения операционной системы или с помощью параметра командной строки. Кроме этого можно подключить индексный файл, в котором перечислены подключаемые скрипты. Подробнее об этом читайте в руководстве разработчика.

Функциональные типы

Общая информация

Функциональный тип — это тип, значениями которого являются методы. Функциональные типы позволяют хранить методы в переменных, передавать их в другие методы как аргументы и возвращать как результат.

На первый взгляд непонятно, зачем это может понадобиться. На «второй» взгляд возникает вопрос: «Можно ли обойтись без функциональных типов?»

Да, без функциональных типов можно обойтись, но с ними можно писать более лаконичный и стройный код, с ними можно реализовывать более универсальные алгоритмы.

Назначение функционального типа проще всего понять на примере. Основной смысл примера заключается в том, что, когда вы попадаете в какой-то метод, исполняется программный код, написанный внутри этого метода. «Снаружи» вы никак не можете повлиять на то, какая последовательность инструкций будет выполнена. Как это было один раз предусмотрено при разработке этого метода, так это и будет.

Теперь представьте, что вам нужно сравнивать два числа, но в зависимости от ситуации. В одном случае нужно понять, делится ли одно число нацело, а в другом случае нужно понять, есть ли в результате деления целые числа. Возможно, в будущем вам понадобится ещё каким-то образом анализировать результат деления одного числа на другое, но вы пока не знаете каким.

Если бы вы не знали о функциональных типах, то для каждого случая вы написали бы отдельный метод. Появился бы третий случай, вы написали бы для него третий метод.

Возможен другой вариант: вы написали бы один метод, но внутри него анализировали бы каждый из случаев. Если это первый случай, то выполняется такой алгоритм. Если это случай второй — то выполняется другой алгоритм. И так далее.

В этой ситуации функциональные типы позволяют написать универсальный метод, имеющий три параметра. В первые два вы будете передавать два числа, которые нужно сравнить. В третий параметр, параметр функционального типа, вы будете передавать тот метод, который нужно использовать для сравнения этих чисел. То есть, грубо говоря, вы будете передавать тот программный код, который должен выполниться. Возвращать ваш универсальный метод будет результат типа `Булево`, означающий удачный или неудачный анализ.

Таким образом, в зависимости от ситуации каждый раз при вызове этого универсального метода вы можете передавать ему разные методы, которыми надо анализировать числа. Появится у вас новая ситуация, требующая другого способа анализа, вы передадите ему новый метод, который подходит для этого случая.

Описание функционального типа

Вы помните, что в «1С:Элементе» всегда нужно указывать тип переменной, параметра или возвращаемого значения метода. Как описывается функциональный тип?

Поскольку значение функционального типа — это метод, то, чтобы описать любой метод, нужно знать типы его параметров и знать тип возвращаемого значения.

Описание функционального типа как раз и содержит типы параметров и тип возвращаемого значения, разделённые *знаком лямбда-операции* `->`.

Вот некоторые примеры описания функционального типа.

```
знч МойМетод: (Число, Число)->Строка
```

В примере выше описано, что переменная **МойМетод** будет содержать метод, имеющий два параметра типа `Число` и возвращающий значение типа `Строка`.

```
знч МойМетод: (Число, Число)->Строка?
```

В примере выше описано, что переменная **МойМетод** будет содержать метод, имеющий два параметра типа `Число` и возвращающий значение составного типа `Строка` или `Неопределено`.

```
знч МойМетод: (Число?)->Строка|Число
```

В примере выше описано, что переменная **МойМетод** будет содержать метод, имеющий один параметр составного типа `Число` или `Неопределено` и возвращающий значение составного типа `Строка` или `Число`.

```
знч МойМетод: ()->Строка
```

В примере выше описано, что переменная **МойМетод** будет содержать метод без параметров, возвращающий значение типа `Строка`.

```
знч МойМетод: (Строка)->ничто
```

В примере выше описано, что переменная **МойМетод** будет содержать метод с одним параметром типа `Строка`, ничего не возвращающий.

Функциональный тип может являться частью составного типа. В этом случае имя функционального типа должно быть заключено в скобки, иначе все, что следует за лямбда-операцией `->`, будет считаться составным типом результата. Например:

```
знч МойМетод: ((Число)->Строка)?
```

В примере выше описано, что переменная **МойМетод** будет содержать либо `Неопределено`, либо метод с одним параметром типа `Число`, возвращающий значение типа `Строка`.

```
знч МойМетод: ((Число)->Строка)|Булево
```

В примере выше описано, что переменная **МойМетод** будет содержать либо `Булево`, либо метод с одним параметром типа `Число`, возвращающий значение типа `Строка`.

Ключевое слово «ничто»

Ключевое слово `ничто` используется как тип возвращаемого значения метода в том случае, когда метод ничего не возвращает.

В описаниях функционального типа это ключевое слово используется обязательно. Например:

```
знч МойМетод: (Строка)->ничто
```

В этом примере описано, что переменная **МойМетод** будет содержать метод с одним параметром типа `Строка`, ничего не возвращающий.

В объявлениях ваших методов использование этого ключевого слова не является обязательным. Вы можете использовать его для удобства чтения кода и для структурного единообразия кода при типизации. Например, следующие примеры объявления метода являются абсолютно равносильными:

```
метод ПредупредитьОПонедельнике()
;

метод ПредупредитьОПонедельнике(): ничто
;
```

Литерал функционального типа

Значения функционального типа могут записываться двумя способами: с помощью лямбда-выражения и с помощью ссылки на метод.

Лямбда-выражение

Литерал функционального типа, записанный с помощью лямбда-выражения, может выглядеть следующим образом:

```
метод Скрипт()
    знч МетодСравнить = (Строка1: Строка, Строка2: Строка) -> Строка1.Длина() < Строка2.Длина()
;
```

В этом примере в переменной **МетодСравнить** находится метод, имеющий два параметра: **Строка1** типа `Строка` и **Строка2** типа `Строка`:

```
(Строка1: Строка, Строка2: Строка)
```

Этот метод получает длину первой строки и сравнивает её с длиной второй строки. Возвращается результат этого сравнения, имеющий тип `Булево`:

```
Строка1.Длина() < Строка2.Длина()
```

Лямбда-выражение — это анонимный метод, то есть метод, описанный в коде скрипта, но не имеющий имени. Обычные методы имеют имя потому, что вы вызываете их из других частей скрипта, чтобы выполнить их код. Лямбда-выражение не имеет имени потому, что код, содержащийся в лямбда-выражении, предназначен для передачи в то место, где он будет выполнен.

Внешне простое лямбда-выражение похоже на описание функционального типа. Сначала в скобках перечислены параметры, затем идёт знак *лямбда-операции* `->`, а потом написан какой-то программный текст:

```
(Строка, Строка)->Булево                                                    // описание функционального типа
(Строка1: Строка, Строка2: Строка) -> Строка1.Длина() < Строка2.Длина()     // лямбда-выражение
```

Не нужно их путать. Во-первых, принято по-разному писать знак лямбда-операции `->`: в описании типа он не обрамляется пробелами, а в лямбда-выражении, наоборот, принято обрамлять его пробелами. Во-вторых, лямбда-выражения могут быть сложнее, о чём вы узнаете далее *(222)*.

Ссылка на метод

Литерал функционального типа из предыдущего примера, записанный с помощью ссылки на метод, может выглядеть следующим образом:

```
метод Скрипт()
    знч МетодСравнить = &СравнитьНаМеньше
;

метод СравнитьНаМеньше(Строка1: Строка, Строка2: Строка): Булево
    возврат Строка1.Длина() < Строка2.Длина()
;
```

Ссылка на метод начинается с амперсанда «&», за которым следует имя метода, объявленного где-то в области видимости. В этом примере в переменной **МетодСравнить** находится метод с именем **СравнитьНаМеньше()**, описанный ниже.

> **Примечание:**
>
> Амперсанд «&» можно вводить с помощью Alt-ввода *(227)*.

Пример: хранение метода в переменной

Значение функционального типа можно присвоить переменной и в дальнейшем обращаться к этой переменной, как к самому методу.

Например, переменной **НаписатьДату** можно присвоить лямбда-выражение, которое получает дату и возвращает её представление.

```
метод Скрипт()
    знч НаписатьДату = (Дата: Дата) -> Дата.Представление("'Сегодня 'дд ММММ гггг' ('дддд')'")
    пер Сообщение = НаписатьДату(ДатаВремя.Сейчас().Дата)
;
```

В результате в переменной **Сообщение** будет значение **"Сегодня 26 июня 2025 (четверг)"**, например.

Этот же пример, записанный с помощью ссылки на метод, будет выглядеть следующим образом:

```
метод Скрипт()
    знч НаписатьДату = &ПредставлениеДаты
    пер Сообщение = НаписатьДату(ДатаВремя.Сейчас().Дата)
;

метод ПредставлениеДаты(Дата: Дата): Строка
```

```
    возврат Дата.Представление("'Сегодня 'дд MMMM гггг' ('дддд')'")
;
```

Пример: ссылка на системный метод

Можно ссылаться не только на собственные методы, но и на системные. Например, повторяя предыдущий пример, можно сослаться прямо на системный метод `Дата.Представление()`:

```
метод Скрипт()
    знч НаписатьДату = &Дата.Представление(Строка)
    пер Сообщение = НаписатьДату(ДатаВремя.Сейчас().Дата, "'Сегодня 'дд MMMM гггг' ('дддд')'")
;
```

В этом случае достаточно после амперсанда указать имя типа и имя метода. Однако если системный метод имеет перегрузки, то, чтобы выбрать правильный вариант, «1С:Элемент» попросит уточнить типы параметров метода, в данном случае — нужен метод с параметром типа `Строка`:

```
&Дата.Представление(Строка)
```

Значение функционального типа, полученное таким образом, **НаписатьДату**, будет иметь первым параметром параметр указанного типа (то есть значение типа `Дата`, к которому нужно применить метод **Представление()**), а далее — такие же параметры, что и системный метод, и такой же тип возврата.

Поэтому первым параметром передаётся дата, которую нужно представить, а вторым параметром — форматная строка.

```
НаписатьДату(ДатаВремя.Сейчас().Дата, "'Сегодня 'дд MMMM гггг' ('дддд')'")
```

Пример: вызов системного метода с параметром функционального типа

Некоторые системные методы имеют параметры функционального типа. Например, метод `Массив.СортироватьПо()` имеет первый параметр функционального типа. В этот параметр нужно передать лямбда-выражение, которое вернёт значение, по которому нужно сортировать массив.

Например, массив **Студенты** состоит из элементов, каждый из которых описан в виде структуры **Студент**. Нужно отсортировать список студентов по полю **Оценка** этой структуры:

```
структура Студент
    знч ФИО: Строка
    знч Оценка: Число
;

метод Скрипт()
    пер Студенты = [новый Студент("Стрельцов",4),
                    новый Студент("Малахова", 5),
                    новый Студент("Соколов", 4),
                    новый Студент("Королев",3),
                    новый Студент("Мальцева", 5)]
    Студенты = Студенты.СортироватьПо(ПарамСтудент -> ПарамСтудент.Оценка, НаправлениеСортировки.ПоУбыванию)
;
```

В результате в массиве **Студенты** будет значение **[{"Малахова":5}, {"Мальцева":5}, {"Стрельцов":4}, {"Соколов":4}, {"Королев":3}]**.

Здесь в методе `Массив.СортироватьПо()`:

- В первом аргументе указано лямбда-выражение, которое возвращает значение поля структуры (**Оценка**), по которому массив нужно отсортировать:

```
ПарамСтудент -> ПарамСтудент.Оценка
```

- Во втором аргументе указан порядок сортировки.

В результате для каждого экземпляра структуры **Студент**, содержащегося в массиве **Студенты**, извлекается значение поля структуры **Оценка**, и массив сортируется по этим значениям.

Пример: передача метода в другой метод

Вот пример, с которого начинается раздел. Нужно сравнивать два числа, но в зависимости от ситуации. В одном случае нужно понять, делится ли одно число нацело, а в другом — нужно понять, есть ли в результате деления целые числа. Возможно, в будущем вам понадобится ещё каким-то образом анализировать результат деления одного числа на другое, но вы пока не знаете каким.

В этой ситуации вы можете написать универсальный метод, **МожноРазделить()**, куда будете передавать два числа и получать результат типа `Булево`, который будет обозначать удачный или неудачный анализ:

```
метод МожноРазделить(Делимое: Число, Делитель: Число, Разделить: (Число, Число)->Булево): Булево
    возврат Разделить(Делимое, Делитель)
;
```

Как именно анализировать результат деления, вы будете передавать в ещё одном параметре этого метода, в параметре функционального типа **Разделить**:

```
Разделить: (Число, Число)->Булево
```

В этот параметр каждый раз при вызове метода вы будете передавать в виде лямбда-выражения ту функцию, которую нужно выполнить:

```
(Число1, Число2) -> Число1 % Число2 == 0
или
(Число1, Число2) -> Число1 % Число2 > 1
```

Итак, в одной ситуации вам нужно понять, делятся ли предметы (кружки) нацело, т.к. один предмет невозможно разделить на части. В другой ситуации нужно понять, достаточно ли предметов (яблок), чтобы каждый получил по целому яблоку:

```
метод Скрипт()

    // Можно разделить - это значит, что без остатка
    // 8 кружек разделить между 4 людьми. Кружку нельзя сломать и разделить на части.
    пер Делимое = 8
    пер Делитель = 4
```

```
    пер Результат = МожноРазделить(Делимое, Делитель, (Число1, Число2) -> Число1 % Число2 == 0)

    // Можно разделить - это значит, что частное больше единицы
    // 10 яблок между 4 людьми. Яблоки маленькие, давать человеку только часть яблока нет смысла.
    Делимое = 10
    Делитель = 4
    Результат = МожноРазделить(Делимое, Делитель, (Число1, Число2) -> Число1 % Число2 > 1)
;

метод МожноРазделить(Делимое: Число, Делитель: Число, Разделить: (Число, Число)->Булево): Булево
    возврат Разделить(Делимое, Делитель)
;
```

В результате в переменной **Результат** в обоих случаях будет значение `Истина`.

Вместо лямбда-выражений можно использовать ссылки на существующие методы. Например, если алгоритмы сравнения уже описаны в методах **МожноРазделитьБезОстатка()** и **МожноРазделитьСЦелойЧастью()**, то вместо лямбда-выражений можно использовать ссылки на эти существующие методы.

```
метод Скрипт()

    // Можно разделить - это значит, что без остатка
    // 8 кружек разделить между 4 людьми. Кружку нельзя сломать и разделить на части.
    пер Делимое = 8
    пер Делитель = 4
    пер Результат = МожноРазделить(Делимое, Делитель, &МожноРазделитьБезОстатка)

    // Можно разделить - это значит, что частное больше единицы
    // 10 яблок между 4 людьми. Яблоки маленькие, давать человеку только часть яблока нет смысла.
    Делимое = 10
    Делитель = 4
    Результат = МожноРазделить(Делимое, Делитель, &МожноРазделитьСЦелойЧастью)
;

метод МожноРазделитьБезОстатка(Делимое: Число, Делитель: Число): Булево
    возврат Делимое % Делитель == 0
;

метод МожноРазделитьСЦелойЧастью(Делимое: Число, Делитель: Число): Булево
    возврат Делимое % Делитель > 1
;

метод МожноРазделить(Делимое: Число, Делитель: Число, Разделить: (Число, Число)->Булево): Булево
    возврат Разделить(Делимое, Делитель)
;
```

Как записывать лямбда-выражения

Часть лямбда-выражения, расположенная после знака лямбда-операции `->`, называется *телом лямбда-выражения*. Тело лямбда-выражения можно записать в двух разных формах: в виде однострочного выражения или в виде последовательности инструкций.

Тело лямбда-выражения, записанное в виде одной строки, вы уже хорошо знаете. Например:

```
метод Скрипт()
    знч МойМетод = (ПарСтрока: Строка) -> ПарСтрока.Длина()
;
```

Если тело содержит от двух до пяти строк, то лучше записывать лямбда-выражение в виде последовательности инструкций. В этом случае лямбда-выражение начинается с ключевого слова `метод` и заканчивается точкой с запятой «;»:

```
метод Скрипт()
    пер ТекущийДеньНедели = ДеньНедели.Суббота
    знч МойМетод = ВыходнойДень(ТекущийДеньНедели,
                                метод(ДеньНедели) ->
                                    выбор ДеньНедели
                                    когда Суббота, Воскресенье
                                        возврат "Сегодня выходной"
                                    иначе
                                        возврат "Будний день"
                                    ;
                                ;
                            )
;

метод ВыходнойДень(ДеньНедели: ДеньНедели, КакойДень: (ДеньНедели)->Строка): Строка
    возврат КакойДень(ДеньНедели)
;
```

Если тело лямбда-выражения больше пяти строк, то лучше записывать тело как обычный метод и использовать ссылку на этот метод:

```
метод Скрипт()
    знч МойМетод = &ОписаниеДня
;

метод ОписаниеДня(ТекущийДеньНедели: ДеньНедели): Строка
    пер СообщениеДня: Строка
    выбор ТекущийДеньНедели
    когда Суббота, Воскресенье
        СообщениеДня = "Сегодня выходной"
    когда Пятница
        СообщениеДня = "Сегодня предвыходной день"
    иначе
        СообщениеДня = "Сегодня будний день"
    ;
    возврат СообщениеДня
;
```

Примечание:

Подробнее о функциональных типах читайте в руководстве разработчика.

Динамическая типизация

Общая информация

Не всегда статическая типизация «1С:Элемента» хороша. Порой в момент написания скрипта вы не знаете, какой тип будет у переменной. Он станет известен только в момент исполнения. Но «1С:Элемент» требует, чтобы вы сразу указали правильный тип. Как быть в такой ситуации?

Учитывая, что все типы «1С:Элемента» образуют иерархию, можно было бы описать тип такой переменной как **Объект?**. Такое описание позволит присвоить переменной значение любого типа, так как тип `Объект` является базовым для любых других типов, кроме `Неопределено`, который также входит в это описание.

Но при таком описании типа переменной вы не сможете, например, вызвать у неё метод `Строка.Длина()`, если предполагаете, что она будет строкового типа.

```
метод  Скрипт()
      пер  МояПеременная:  Объект?
      пер  Результат  =  МояПеременная.Длина()
;
```

В результате компилятор сообщит вам об ошибке **Неизвестный метод "Объект.Длина"**.

Для решения этой проблемы в «1С:Элементе» существует тип `неизвестно`.

Тип «неизвестно»

Тип `неизвестно` существует только во время компиляции исходного текста скрипта. Его назначение — отключить проверку типов компилятора. Все проверки будут выполняться во время исполнения, когда станет ясно, значение какого типа размещено в переменной, объявленной с типом `неизвестно`.

Применение этого типа позволяет вам, например, написать и исполнить предыдущий пример без ошибок:

```
метод  Скрипт()
      пер  МояПеременная:  неизвестно
      МояПеременная  =  "тест"
      пер  Результат  =  МояПеременная.Длина()
;
```

В результате в переменной **Результат** будет значение **4**.

Raw-типы

Raw-типы — это обобщённые типы, для которых не указан тип параметра типа. Для того чтобы описать такой тип, можно использовать тип `неизвестно`. Тогда на этапе компиляции «1С:Элемент» не будет контролировать, какого именно типа элементы содержатся в обобщённом типе.

Это удобно, например, когда вы пишете универсальный метод, обрабатывающий переданный массив, и не знаете заранее, элементы какого типа будут в переданном массиве. В этом случае вы можете описать тип параметра как **МойМассив: Массив<неизвестно>**, а в методе анализировать переданный массив и выполнять методы, соответствующие типу его элементов. Например:

```
метод Скрипт()
    МойМетод(["первый", "второй", "третий"])
    МойМетод([ДатаВремя.Сейчас().Дата] )
;

метод МойМетод(МойМассив: Массив<неизвестно>)
    пер Результат: неизвестно
    выбор МойМассив[0].ПолучитьТип()
    когда Тип<Строка>
        Результат = МойМассив[0].Длина()
    когда Тип<Дата>
        Результат = МойМассив[0].ДеньНедели()
    ;
;
```

В результате в переменной Результат в первый раз будет значение **6**, а во второй — **Четверг**, например.

Механизм отражения

Механизм отражения полезен для написания универсальных алгоритмов. Например, в ваш метод могут быть переданы экземпляры самых разных типов. Вы не знаете, как называются свойства этих экземпляров, какие они имеют значения. Но вам нужно каким-то образом сериализовать эти данные в текстовый файл в виде строк **имя-свойства, значение-свойства**.

Для того чтобы выполнить эту задачу, вы можете получить тип этого экземпляра и от этого типа получить коллекцию его свойств — **Экз.ПолучитьТип().ПолучитьСвойства()**. Эту коллекцию можно обойти в цикле, и для каждого свойства узнать его имя и значение — **Свойство.Имя + "," + Свойство.Получить(Экз)**. Например:

```
структура Товар
    обз знч Наименование: Строка
    пер Артикул: Строка
    пер Цена: Число = 1
;

метод Скрипт()
    пер Экз = новый Товар("Холодильник", "M50", 5300)

    для Свойство из Экз.ПолучитьТип().ПолучитьСвойства()
        Консоль.Записать(Свойство.Имя + "," + Свойство.Получить(Экз))
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Цена,5300
Артикул,M50
Наименование,Холодильник
```

Механизм отражения позволяет выполнять и другие действия с экземплярами: анализировать обобщённые типы, создавать экземпляры. Подробнее читайте об этом в руководстве разработчика.

Особенности системных методов и типов

Обобщённые системные методы

Раньше вы уже познакомились с коллекциями (массив, соответствие, множество) и знаете, что они описываются обобщёнными типами данных *(108)*. Обобщённые типы позволяют типизировать элементы коллекций, то есть сказать, что это не просто массив, а массив, элементами которого являются строки: **Массив<Строка>**.

Следуя этому же принципу, в «1С:Элементе» есть обобщённые системные методы, то есть методы, для которых нужно указывать тип аргумента, с которым вызывается этот метод.

При вызове системного обобщённого метода тип-параметр указывается в угловых скобках после имени метода без пробела. Значение, возвращаемое методом, может быть связано с типом параметра метода.

Примером системного обобщённого метода может служить метод глобального контекста `Макс()`. Он возвращает максимальное значение из переданных ему параметров. Вызов этого метода может выглядеть так.

```
метод Скрипт()
    пер Максимум = Макс<Число>(11, 7, 13)
;
```

В результате в переменной **Максимум** будет значение **13**.

Это простейший пример, в котором сравниваемые значения написаны в явном виде литералами. Но если вместо них будут переменные, то «1С:Элемент» проконтролирует, что их тип должен быть `Число`. Это же касается и возвращаемого значения.

Тип-параметр обобщённого системного метода может быть составным. Например:

```
метод Скрипт()
    пер Максимум = Макс<Строка|Число>("90", "66", "82")
;
```

В результате в переменной **Максимум** будет значение **"90"**.

Тип-параметр можно не указывать при вызове системного обобщённого метода, если он может быть выведен из контекста. Например:

```
метод Скрипт()
    пер Максимум = Макс("сто", "три", "двадцать")
;
```

В результате в переменной **Максимум** будет значение **"три"**.

Статические системные методы

Когда вы знакомились со структурами и перечислениями, вы узнали, что у них могут быть статические методы *(144)*. Статические методы принадлежат типу, а не экземпляру. Для выполнения статического метода не нужен конкретный экземпляр, достаточно указать имя типа и через точку от него имя статического

метода. Это просто такая форма записи обращения к статическому методу. При этом экземпляр не создаётся.

Статические методы существуют и у стандартных типов «1С:Элемента». Например, это уже известный вам метод `ДатаВремя.Сейчас()`. Также существует метод `ЧасовойПояс.Текущий()` и другие.

Типы-одиночки

Типы-одиночки — это типы, у которых существует единственный экземпляр. По этой причине у них нет конструктора. Чтобы обратиться к экземпляру типа-одиночки, нужно просто написать имя типа.

Например, есть хорошо известный вам тип-одиночка `Консоль`. У него есть метод `Записать()`. Вызов этого метода будет выглядеть так:

```
Консоль.Записать("Сообщение")
```

Обратите внимание: такая запись очень похожа на вызов статического системного метода, например `ДатаВремя.Сейчас()`:

```
ДатаВремя.Сейчас()
```

Но это только внешнее сходство. **ДатаВремя.** не создаёт экземпляр, это просто такой способ записи. А вот **Консоль.** полноценно создаёт экземпляр типа `Консоль`.

Чтобы убедиться, вы можете написать вместо **Консоль.Записать("Сообщение")** следующий код:

```
метод  Скрипт()
      пер  Экземпляр  =  Консоль
      Экземпляр.Записать("Сообщение")
;
```

Этот код у вас прекрасно отработает. Но вы не сможете написать аналогичный код для `ДатаВремя`, потому что `ДатаВремя` — это не тип-одиночка и запись **ДатаВремя** не создаёт экземпляр этого типа.

Есть и другие типы-одиночки, например `Символы` или `Файлы`. В справке по стандартной библиотеке они помечены как **Тип-одиночка**.

Ввод английских символов без переключения раскладки клавиатуры

В «1С:Элементе» можно вводить символы из английской раскладки, не переключаясь на неё, — так называемый *Alt-ввод*.

Для этого во всех случаях, где это возможно, используется одно и то же правило. Чтобы ввести английский символ, удерживайте клавишу **Alt** и нажмите ту клавишу, на которой находится нужный вам английский символ.

Такая возможность реализована не для всех английских символов, а только для тех, которые используются в «1С:Элементе». Например:

- Вертикальная черта «|» — **Alt + **. Вертикальная черта используется при перечислении типов *(152)*:

```
пер ПеременнаяСоставногоТипа: Число|Булево = 3
```

- Фигурные скобки «{» и «}» — **Alt + 9** и **Alt + 0**. Цифры набирайте на основной клавиатуре. Фигурные скобки используются при включении в строковый литерал вычисляемых выражений *(167)*:

```
пер ТекстСообщения = "Объем %{100*50*70/2} литров"
```

- Угловые скобки «<» и «>» — **Alt + б** и **Alt + ю**. Угловые скобки используются в логических выражениях *(52)*:

```
если Счетчик < 7
    возврат
;
```

- Квадратные скобки «[» и «]» — **Alt + x** и **Alt + ъ**. Квадратные скобки используются в литерале массива *(112)*:

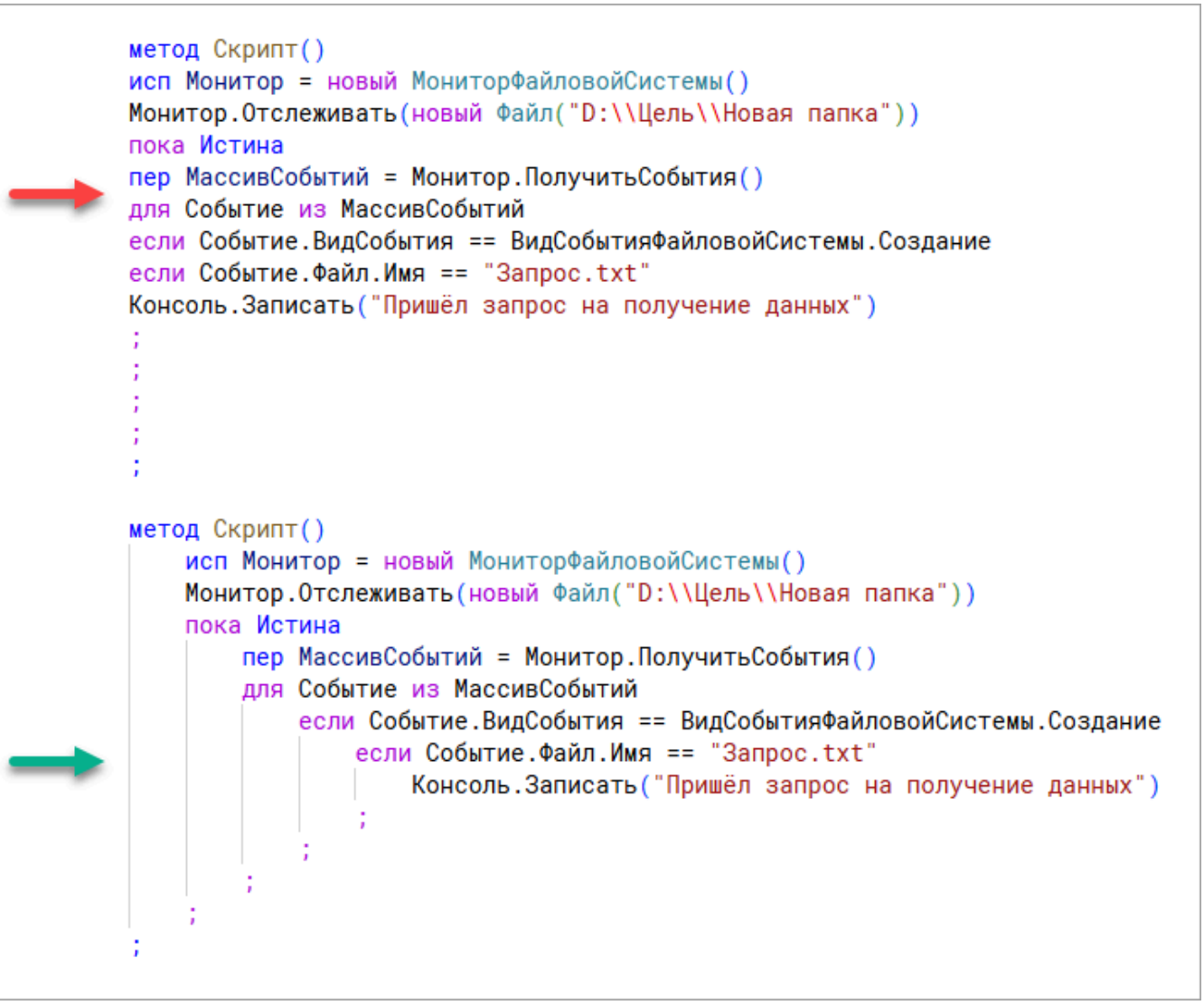
```
пер МассивЧисел = <Число>[0, 1, 2]
```

- Кроме перечисленных символов есть и другие, которые вы можете ввести аналогичным образом:
  - знак доллара «$» — **Alt + 4**, используется в выражении интерполяции *(167)*;
  - амперсанд «&» — **Alt + 7**, используется в ссылке на метод *(219)*;
  - эт (собака) «@» — **Alt + 2**, используется в аннотациях *(213)*;
  - апостроф «'» — **Alt + э**, используется в регулярных выражениях *(188)*;
  - косая черта «/» — **Alt + .**

Рекомендации по написанию кода

Синтаксический отступ

Блоки вложенных инструкций, находящиеся на одном уровне, выделяйте одним синтаксическим отступом. Один синтаксический отступ составляет 4 пробела. Не используйте символы табуляции.



```
метод Скрипт()
исп Монитор = новый МониторФайловойСистемы()
Монитор.Отслеживать(новый Файл("D:\\Цель\\Новая папка"))
пока Истина
пер МассивСобытий = Монитор.ПолучитьСобытия()
для Событие из МассивСобытий
если Событие.ВидСобытия == ВидСобытияФайловойСистемы.Создание
если Событие.Файл.Имя == "Запрос.txt"
Консоль.Записать("Пришёл запрос на получение данных")
;
;
;
;
;

метод Скрипт()
    исп Монитор = новый МониторФайловойСистемы()
    Монитор.Отслеживать(новый Файл("D:\\Цель\\Новая папка"))
    пока Истина
        пер МассивСобытий = Монитор.ПолучитьСобытия()
        для Событие из МассивСобытий
            если Событие.ВидСобытия == ВидСобытияФайловойСистемы.Создание
                если Событие.Файл.Имя == "Запрос.txt"
                    Консоль.Записать("Пришёл запрос на получение данных")
                ;
            ;
        ;
    ;
;
```

Длина строки

Максимальная длина строки — 120 символов. Можно делать строки большей длины, если разбиение таких строк на части по 120 символов ухудшает их чтение и понимание.

Пустые строки

Пустые строки можете использовать по желанию, допустимы оба варианта:

```
метод Скрипт()
    исп Монитор = новый МониторФайловойСистемы()
    Монитор.Отслеживать(новый Файл("D:\\Цель\\Новая папка"))
    пока Истина
        пер МассивСобытий = Монитор.ПолучитьСобытия()
        для Событие из МассивСобытий
            если Событие.ВидСобытия == ВидСобытияФайловойСистемы.Создание
                если Событие.Файл.Имя == "Запрос.txt"
                    Консоль.Записать("Пришёл запрос на получение данных")
                ;
            ;
        ;
    ;
;

метод Скрипт()

    исп Монитор = новый МониторФайловойСистемы()
    Монитор.Отслеживать(новый Файл("D:\\Цель\\Новая папка"))

    пока Истина
        пер МассивСобытий = Монитор.ПолучитьСобытия()

        для Событие из МассивСобытий
            если Событие.ВидСобытия == ВидСобытияФайловойСистемы.Создание
                если Событие.Файл.Имя == "Запрос.txt"

                    Консоль.Записать("Пришёл запрос на получение данных")
                ;
            ;
        ;
    ;
;
```

Составные инструкции

Составные инструкции завершайте точкой с запятой «;», пишите её на отдельной строке с тем же синтаксическим отступом, что и сама инструкция.

```
для Счетчик = 1 по 5
    ПервыйМетод()
    ВторойМетод()
    ;


для Счетчик = 1 по 5
    ПервыйМетод()
    ВторойМетод()
;
```

```
если ПеременнаяПервая > 5
    ПервыйМетод()
    ВторойМетод()
иначе
    ТретийМетод()
    ;

если ПеременнаяПервая > 5
    ПервыйМетод()
    ВторойМетод()
иначе
    ТретийМетод()
;
```

```
метод МойМетод(): Булево
    пер Переменная1 = Истина
    возврат Переменная1
    ;

метод МойМетод(): Булево
    пер Переменная1 = Истина
    возврат Переменная1
;
```

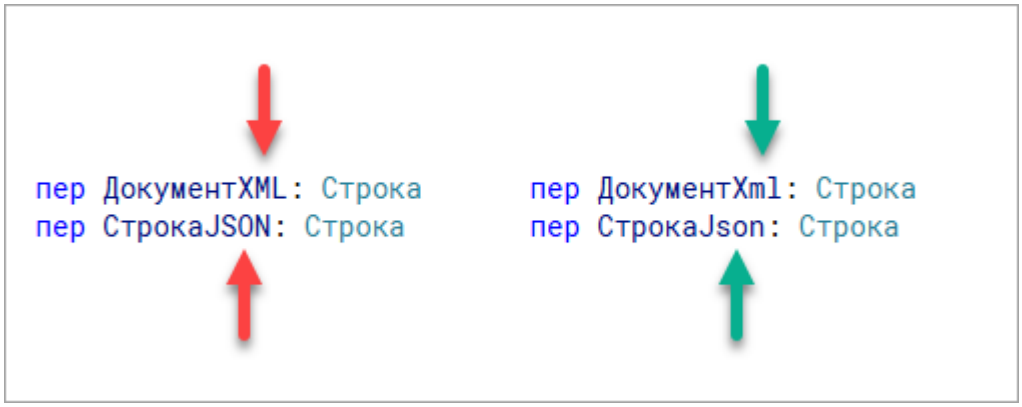
Имена

Все имена, которые вы используете для переменных, типов и методов, пишите в стиле CamelCase. Если имя состоит из нескольких слов, то пишите их слитно без пробелов, при этом каждое слово пишите с прописной буквы. Если имя состоит из одного слова, тоже пишите его с прописной буквы.

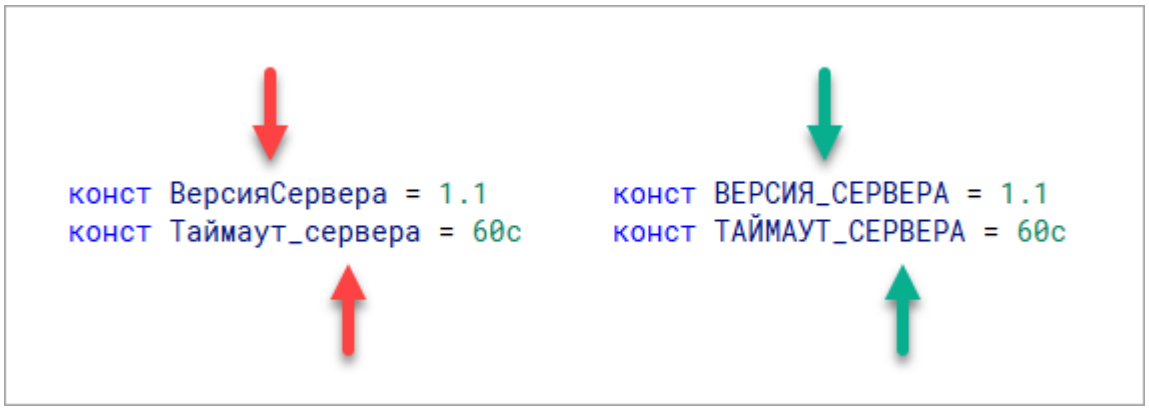
```
перечисление Степень_Важности
    высокая
;

структура входящееСообщение
;

метод версия()
;
```

```
перечисление СтепеньВажности
    Высокая
;

структура ВходящееСообщение
;

метод Версия()
;
```

В аббревиатурах только первую букву пишите прописной, остальные — строчные: **Xml**, **Json**, **Uuid**.

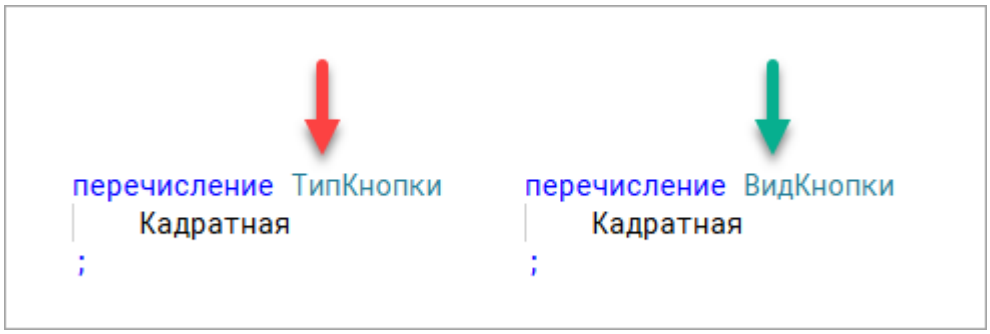


Имена констант пишите заглавными буквами с подчёркиванием.



Типы исключений называйте с префиксом **Исключение**.

В именах перечислений используйте слово **Вид** вместо **Тип**:



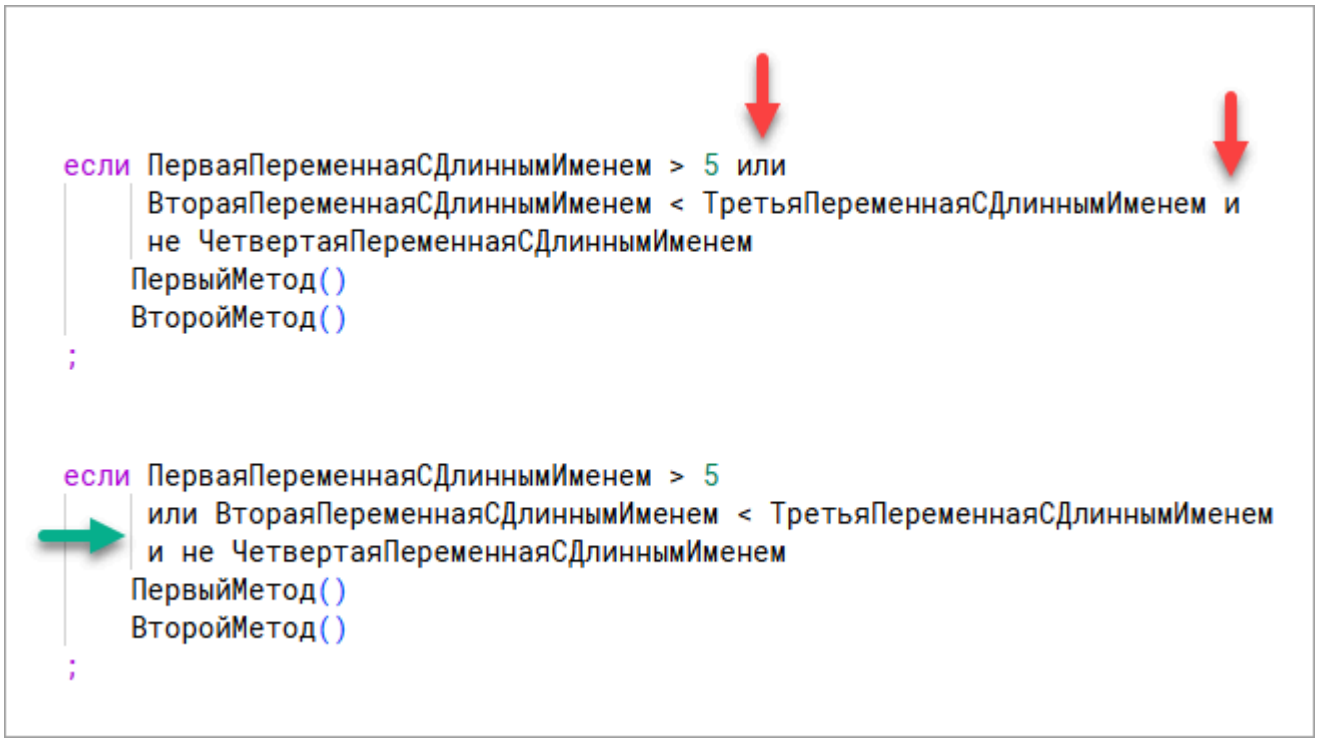
Перенос выражений

Переносите выражения в тех случаях, когда:

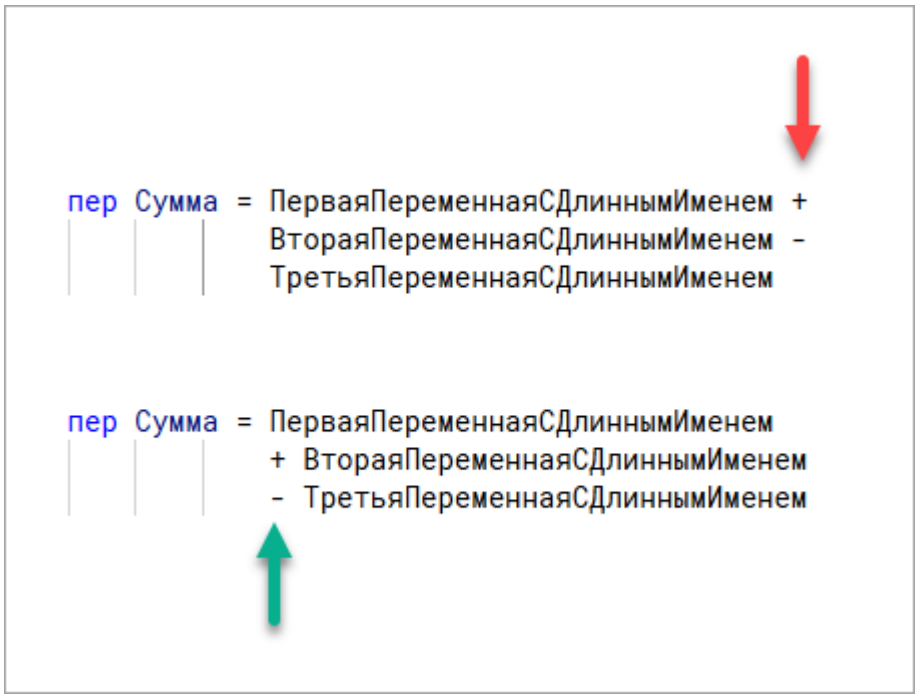
- инструкция, содержащая выражение, превышает максимальную длину строки;
- перенос выражения улучшает понимание инструкции.

В каждой перенесённой строке может содержаться одна или несколько операций.

Операции пишите **в начале** перенесённых строк.

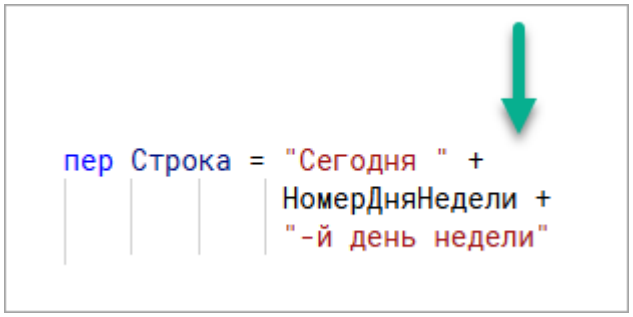


```
если ПерваяПеременнаяСДлиннымИменем > 5 или
     ВтораяПеременнаяСДлиннымИменем < ТретьяПеременнаяСДлиннымИменем и
     не ЧетвертаяПеременнаяСДлиннымИменем
    ПервыйМетод()
    ВторойМетод()
;

если ПерваяПеременнаяСДлиннымИменем > 5
     или ВтораяПеременнаяСДлиннымИменем < ТретьяПеременнаяСДлиннымИменем
     и не ЧетвертаяПеременнаяСДлиннымИменем
    ПервыйМетод()
    ВторойМетод()
;
```



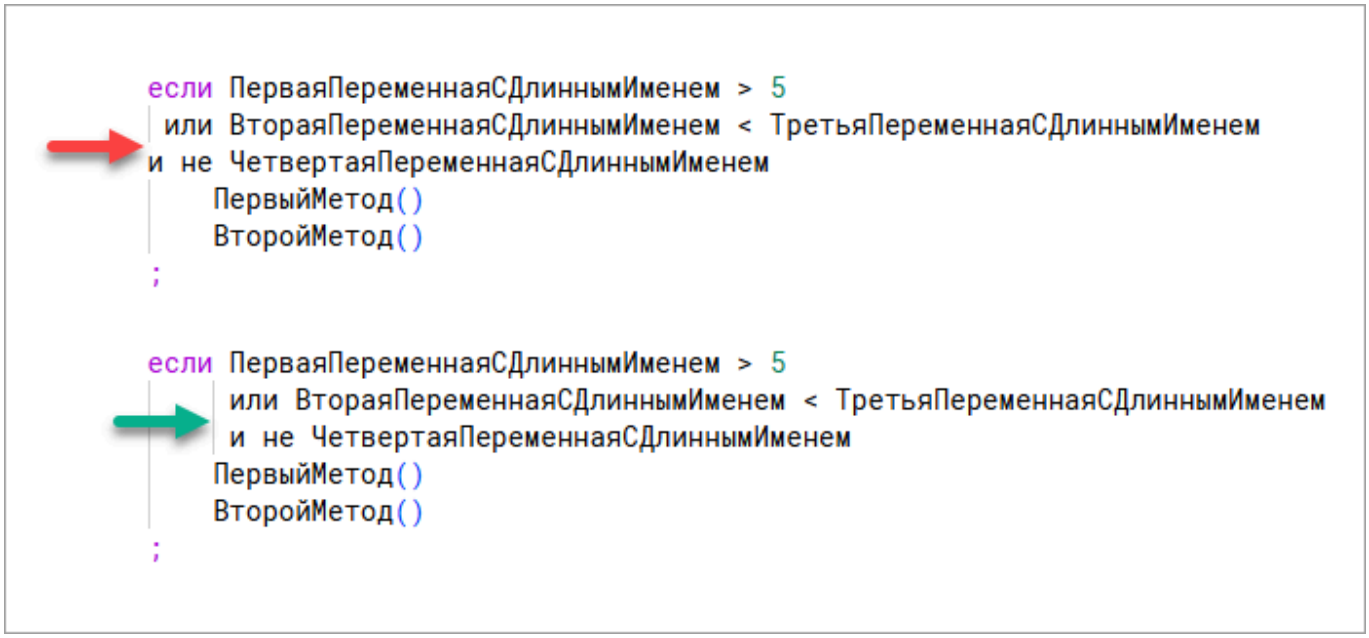
```
пер Сумма = ПерваяПеременнаяСДлиннымИменем +
            ВтораяПеременнаяСДлиннымИменем -
            ТретьяПеременнаяСДлиннымИменем

пер Сумма = ПерваяПеременнаяСДлиннымИменем
          + ВтораяПеременнаяСДлиннымИменем
          - ТретьяПеременнаяСДлиннымИменем
```

При **конкатенации** строк **можно** писать **операцию в конце** строки:

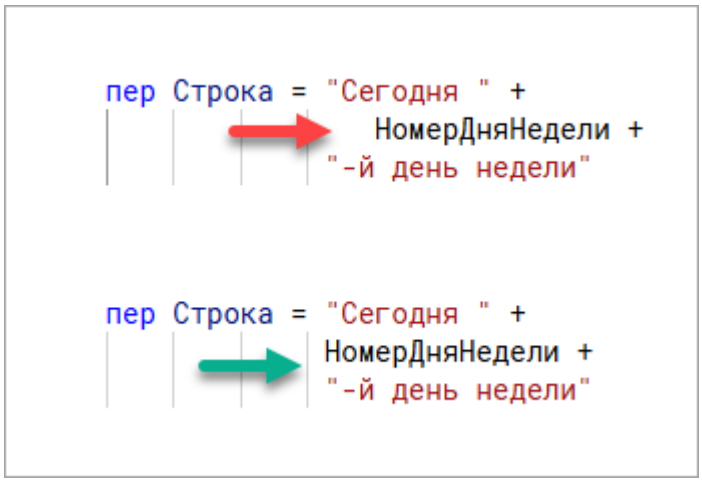


```
пер Строка = "Сегодня " +
             НомерДняНедели +
             "-й день недели"
```

Перенесённые строки можно **выравнивать** (способ 1) **по началу первого операнда**.

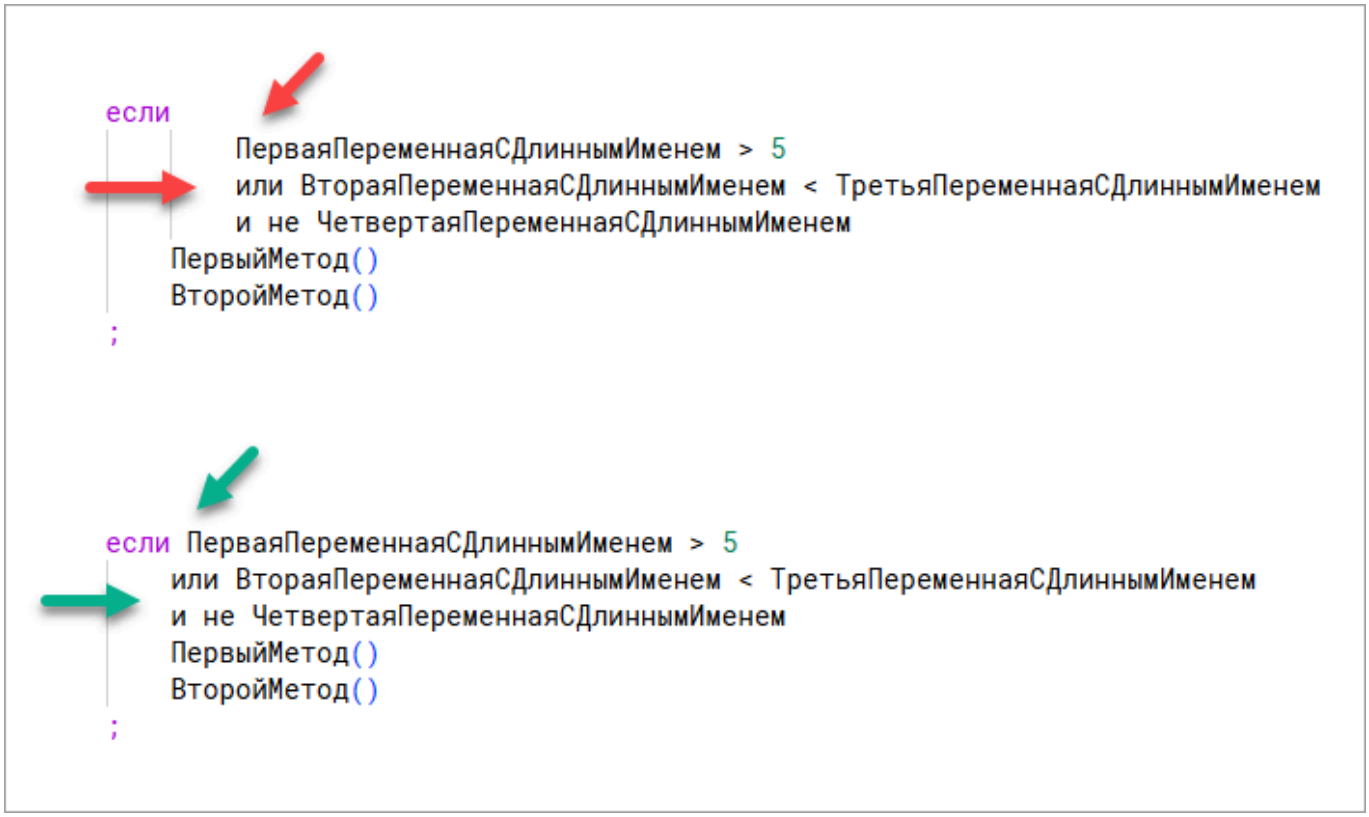


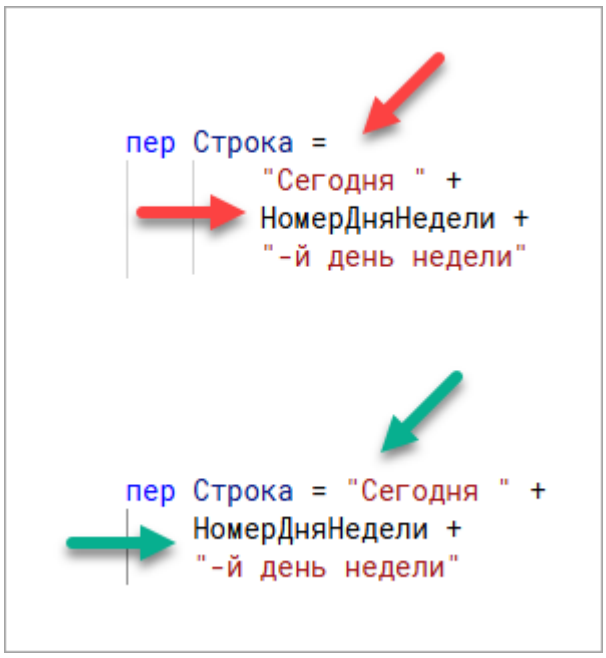
```
если ПерваяПеременнаяСДлиннымИменем > 5
 или ВтораяПеременнаяСДлиннымИменем < ТретьяПеременнаяСДлиннымИменем
и не ЧетвертаяПеременнаяСДлиннымИменем
    ПервыйМетод()
    ВторойМетод()
;

если ПерваяПеременнаяСДлиннымИменем > 5
    или ВтораяПеременнаяСДлиннымИменем < ТретьяПеременнаяСДлиннымИменем
    и не ЧетвертаяПеременнаяСДлиннымИменем
    ПервыйМетод()
    ВторойМетод()
;
```



```
пер Строка = "Сегодня " +
               НомерДняНедели +
             "-й день недели"

пер Строка = "Сегодня " +
             НомерДняНедели +
             "-й день недели"
```

Перенесённые строки можно **выравнивать** (способ 2) **одним синтаксическим отступом**.





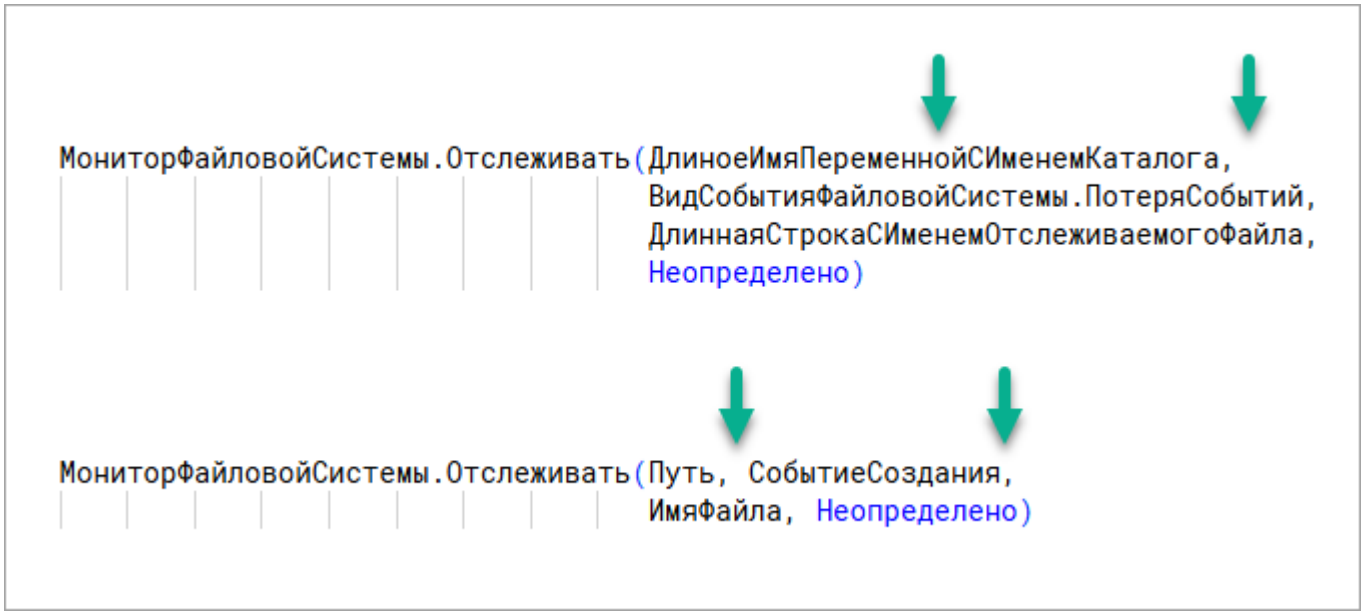
Перенос параметров и литералов коллекций

Параметры методов переносятся в тех случаях, когда:

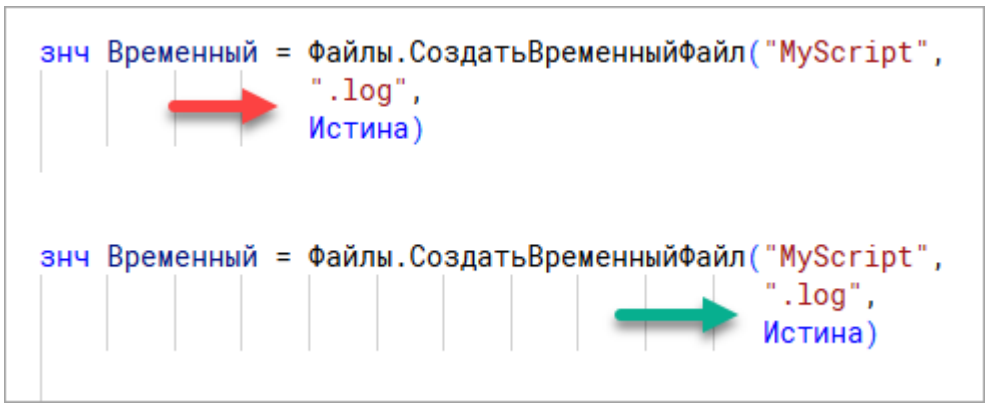
- вызов или объявление метода превышает максимальную длину строки;
- перенос параметров улучшает понимание инструкции.

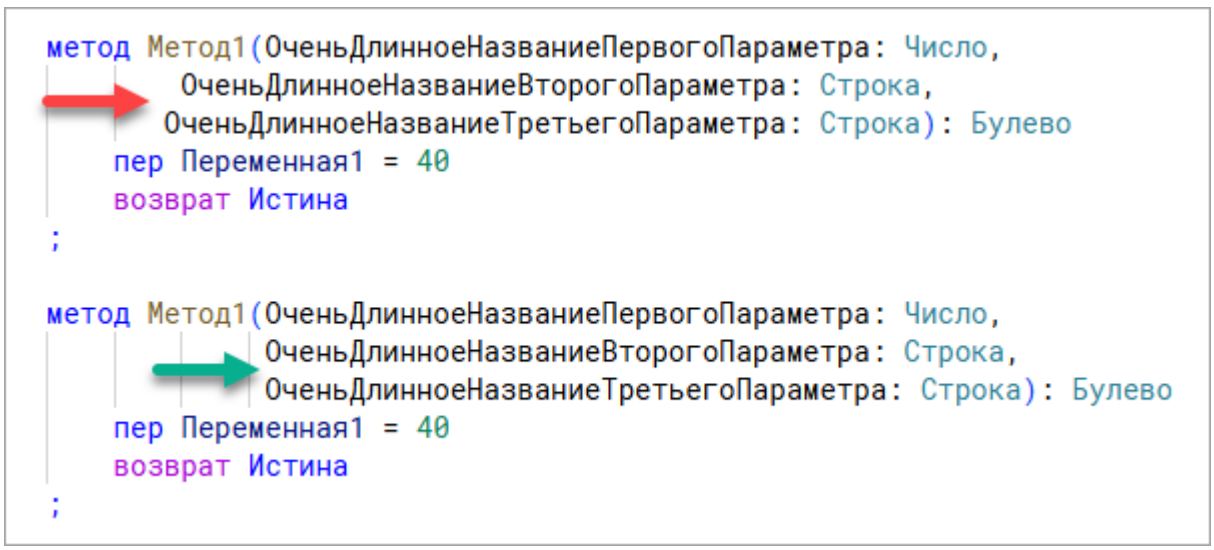
В каждой перенесённой строке может содержаться один или несколько параметров.

Запятые, разделяющие параметры, пишите **в конце** строк:



Перенесённые параметры можно **выравнивать** (способ 1) **по началу первого параметра**. В этом случае закрывающую скобку пишите в конце последней перенесённой строки.





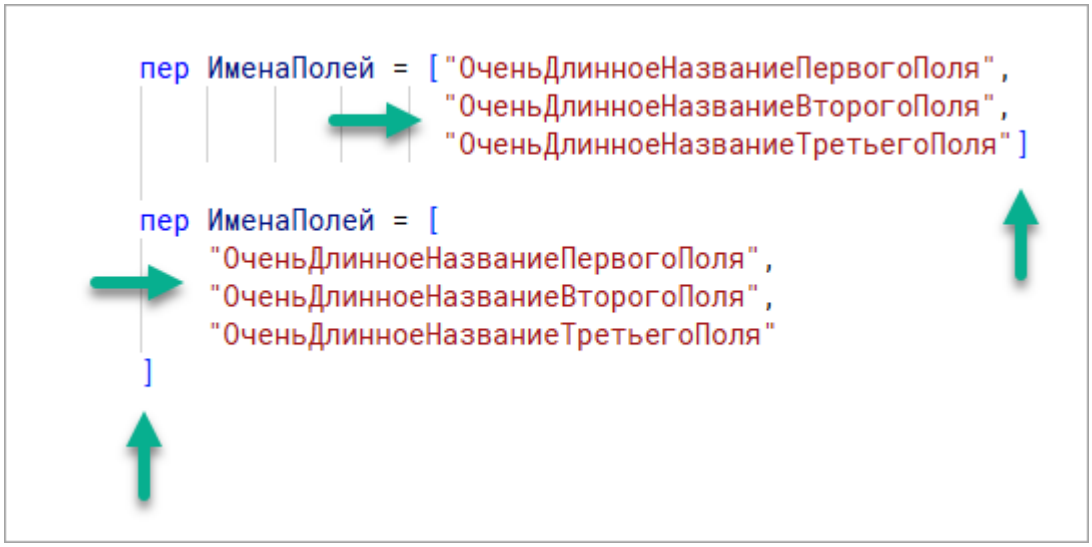
Перенесённые параметры можно **выравнивать** (способ 2) **одним синтаксическим отступом**. В этом случае:

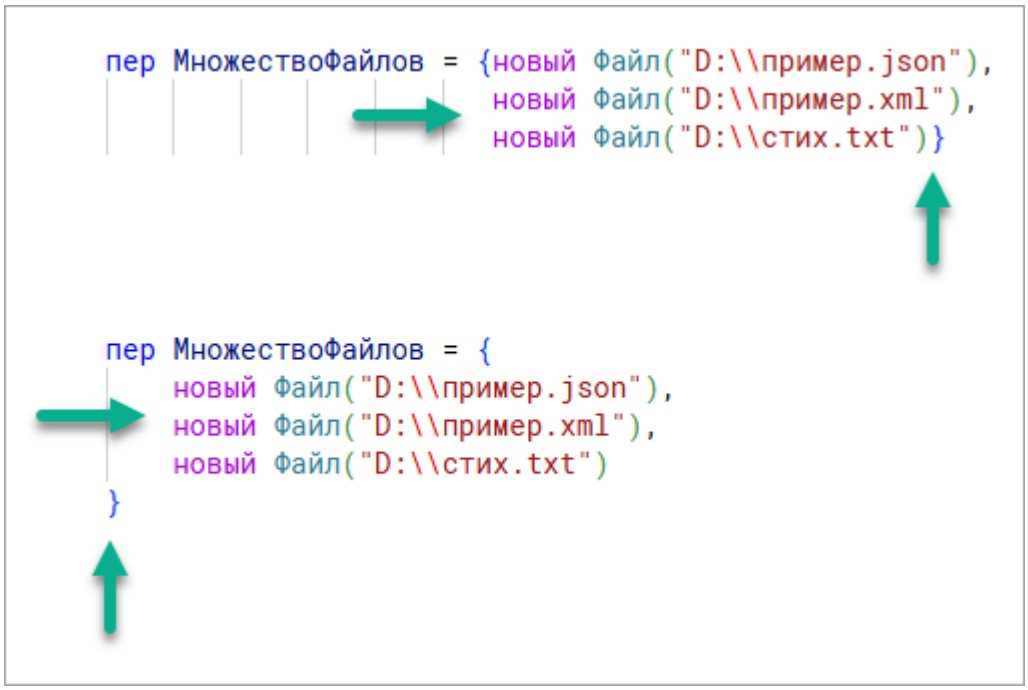
- переносите все параметры, начиная с первого;
- закрывающую скобку пишите:
  - либо на отдельной строке с отступом, соответствующим отступу всей инструкции;
  - либо в конце последней перенесённой строки.

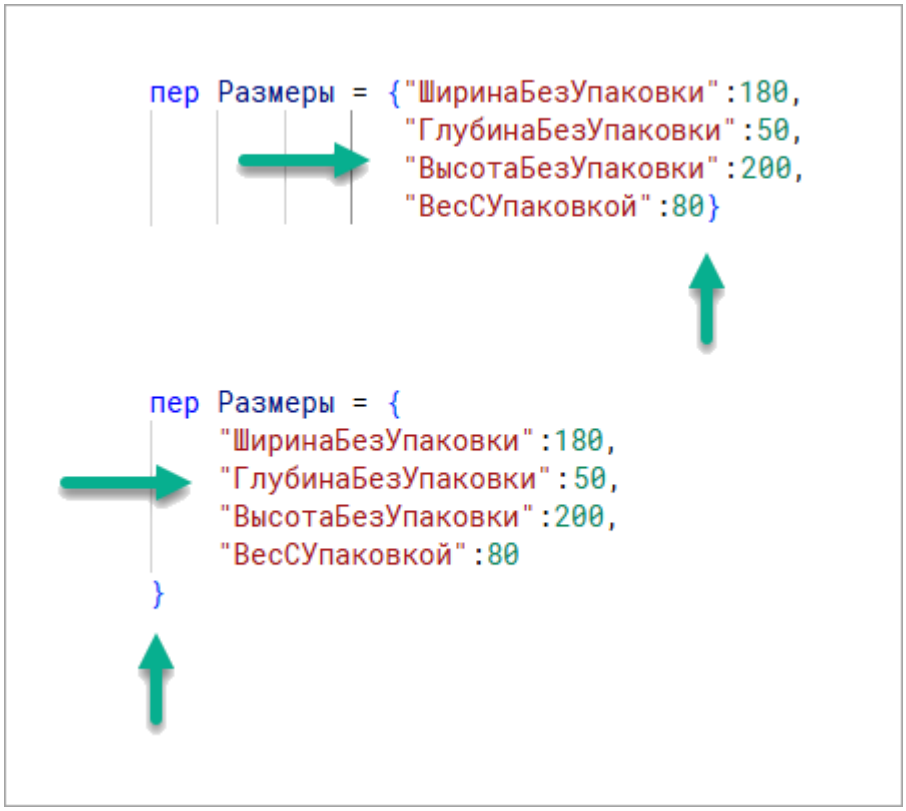
```
знч Временный = Файлы.СоздатьВременныйФайл(
                    "ОченьДлинноеНазваниеПрефикса",
                    ".ОченьДлинноеНазваниеРасширенияФайла",
                    Истина
                )

знч Временный = Файлы.СоздатьВременныйФайл(
                    "ОченьДлинноеНазваниеПрефикса",
                    ".ОченьДлинноеНазваниеРасширенияФайла",
                    Истина)
```

```
    метод Метод1(
        ОченьДлинноеНазваниеПервогоПараметра: Число,
        ОченьДлинноеНазваниеВторогоПараметра: Строка,
        ОченьДлинноеНазваниеТретьегоПараметра: Строка
    ): Булево
        пер Переменная1 = 40
        возврат Истина
    ;

    метод Метод1(
        ОченьДлинноеНазваниеПервогоПараметра: Число,
        ОченьДлинноеНазваниеВторогоПараметра: Строка,
        ОченьДлинноеНазваниеТретьегоПараметра: Строка): Булево
        пер Переменная1 = 40
        возврат Истина
    ;
```

Литералы коллекций переносите по тем же правилам, что и параметры методов:

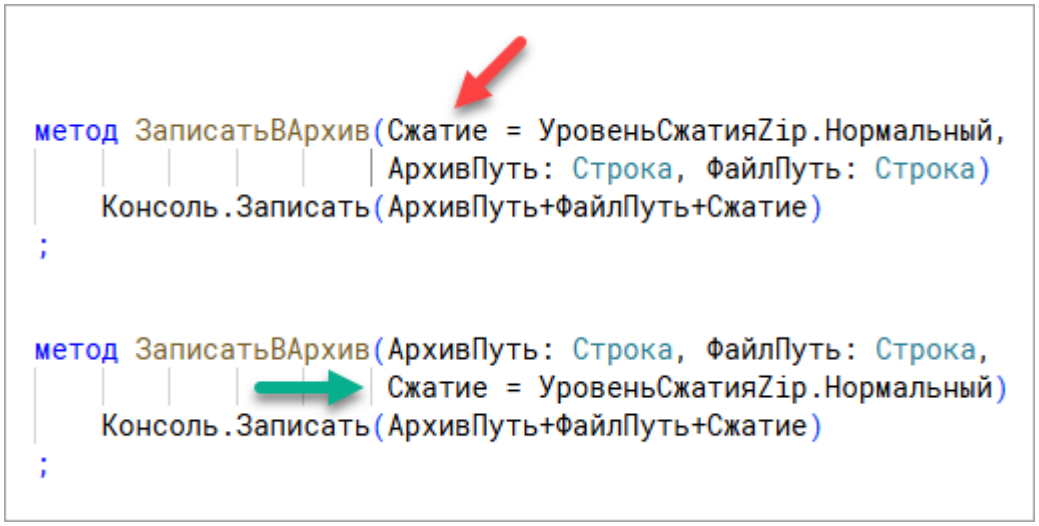

пер ИменаПолей = ["ОченьДлинноеНазваниеПервогоПоля",
"ОченьДлинноеНазваниеВторогоПоля",
"ОченьДлинноеНазваниеТретьегоПоля"]
пер ИменаПолей = [
"ОченьДлинноеНазваниеПервогоПоля",
"ОченьДлинноеНазваниеВторогоПоля",
"ОченьДлинноеНазваниеТретьегоПоля"
]


пер МножествоФайлов = {новый Файл("D:\\пример.json"),
новый Файл("D:\\пример.xml"),
новый Файл("D:\\стих.txt")}
пер МножествоФайлов = {
новый Файл("D:\\пример.json"),
новый Файл("D:\\пример.xml"),
новый Файл("D:\\стих.txt")
}



Объявления методов

Параметры со значениями по умолчанию описывайте после параметров, которые таких значений не имеют.

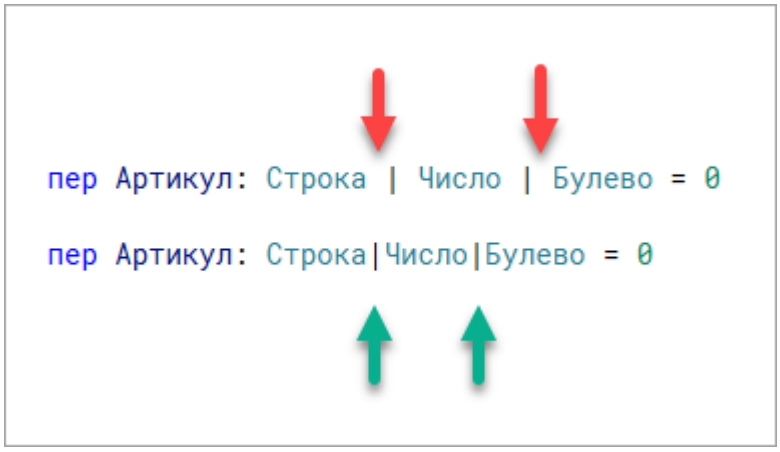


Синтаксис описания типа

Отделяйте тип от имени двоеточием и пробелом. Перед двоеточием пробел не нужен.

Описание составного типа

Не ставьте пробелы вокруг вертикальной черты «|»:

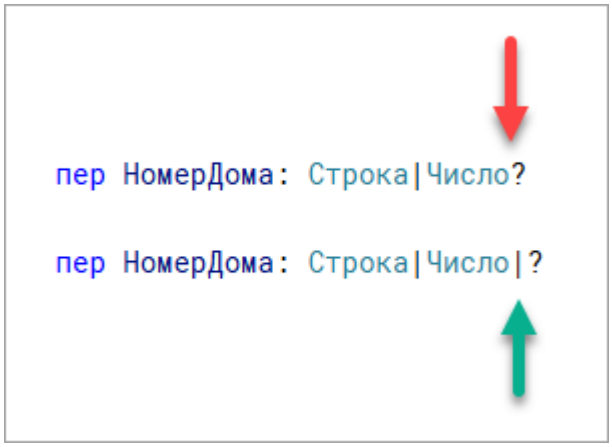


«Неопределено» в описании типа

Тип `Неопределено` в списке типов явно не указывается: вместо него используйте вопросительный знак **?**.

Если типов два, пишите вопросительный знак слитно с предыдущим типом.

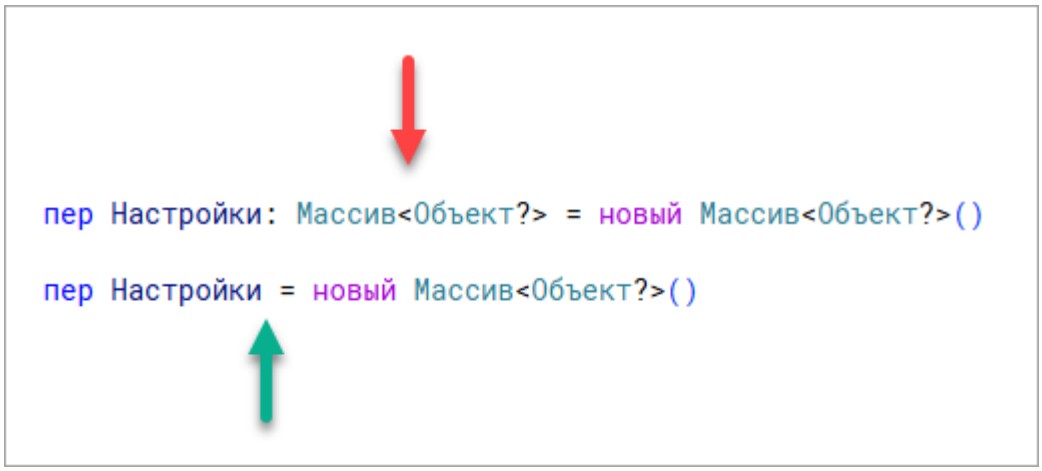
Если типов больше двух, пишите вопросительный знак через вертикальную черту «|».



Инициализация

Не указывайте тип переменной, если она инициализируется литералом.

Не указывайте тип переменной, если она инициализируется конструктором.



Для начального заполнения коллекций используйте литералы, а не заполняйте их с помощью методов.

```
пер МассивПолей: Массив<Строка>
МассивПолей.Добавить("ПервоеПоле")
МассивПолей.Добавить("ВтороеПоле")
МассивПолей.Добавить("ТретьеПоле")

пер МассивПолей = ["ПервоеПоле", "ВтороеПоле", "ТретьеПоле"]
```

```
пер МножествоФайлов: Множество<Файл>
МножествоФайлов.Добавить(новый Файл("D:\\пример.json"))
МножествоФайлов.Добавить(новый Файл("D:\\пример.xml"))
МножествоФайлов.Добавить(новый Файл("D:\\стих.txt"))

пер МножествоФайлов = {новый Файл("D:\\пример.json"),
                       новый Файл("D:\\пример.xml"),
                       новый Файл("D:\\стих.txt")}
```

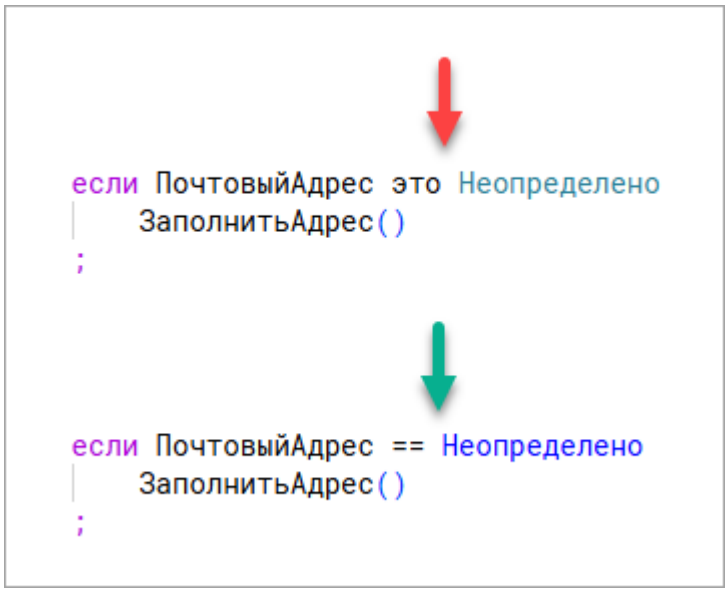
```
пер Размеры: Соответствие<Строка, Число>
Размеры.Вставить("Ширина", 180)
Размеры.Вставить("Глубина", 50)
Размеры.Вставить("Высота", 200)
Размеры.Вставить("Вес", 80)

пер Размеры = {"Ширина":180, "Глубина":50, "Высота":200, "Вес":80}
```

Проверка логических значений

В инструкции `если` не сравнивайте булевы значения на равенство с литералами `Истина` или `Ложь`.

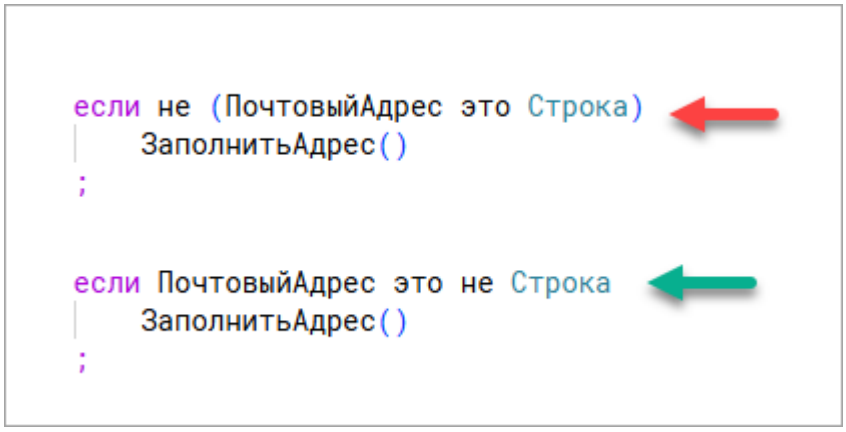
Принадлежность к типу «Неопределено»

Тип `Неопределено` имеет единственное значение `Неопределено`. Чтобы узнать, принадлежит ли ваше значение типу `Неопределено`, вместо операции проверки соответствия типу `это` сравнивайте ваше значение с литералом `Неопределено`.



Операция «это»

Вместо отрицания результата проверки используйте операцию проверки с отрицанием:


если не (ПочтовыйАдрес это Строка)
ЗаполнитьАдрес()
;
если ПочтовыйАдрес это не Строка
ЗаполнитьАдрес()
;

Глава 7. Прикладные возможности

Работа с файлами

Проверить, что файл или каталог существуют

Большинство операций с файлами и файловой системой начинаются с того, что нужно убедиться в существовании файла или каталога. Это можно сделать двумя способами, и выбор способа зависит от того, в какой части алгоритма вы это делаете.

Если вы хотите убедиться в существовании файла или каталога, чтобы передать их в метод, который что-то с ними будет делать, — это один способ. То есть вы хотите, чтобы этот метод гарантированно не имел проблем с отсутствием файла или каталога.

Если вы просто выполняете файловые операции и в процессе их выполнения файл или каталог могут отсутствовать, а вы готовы обработать эту ситуацию прямо здесь, — тогда это другой способ.

В первом случае вам поможет метод `Файл.Существует()`. Во втором случае вам поможет перехват исключения *(200)* при выполнении метода `Файлы.Найти()`. Обратите внимание, что речь идёт о разных типах: `Файл` и `Файлы`.

Использование метода `Файл.Существует()`

Чтобы проверить существование файла или каталога, сконструируйте экземпляр `Файл` и вызовите у него метод `Существует()`. Например:

```
метод Скрипт()
    пер ИсточникКаталог = новый Файл("D:\\Источник")
    если не ИсточникКаталог.Существует()
        Консоль.Записать("Нет каталога-источника")
    ;
;
```

Если каталог **D:\Источник** отсутствует, в терминал будет выведено следующее сообщение:

```
Нет каталога-источника
```

Обратите внимание, что в этом случае экземпляр `Файл` нужен вам только для того, чтобы вызвать его метод `Существует()`. Поэтому вы можете не помещать его в отдельную переменную **ИсточникКаталог**, а вызвать метод `Существует()` сразу от конструктора. Это сделает ваш код более компактным:

```
метод Скрипт()
    если не новый Файл("D:\\Источник").Существует()
        Консоль.Записать("Нет каталога-источника")
    ;
;
```

В дальнейших примерах вы будете использовать именно такую запись.

Перехват исключения в методе `Файлы.Найти()`

Если вы выполняете файловые операции, то все они, как правило, начинаются с того, что нужно найти, отобрать, какие-то файлы, которые вы будете копировать, удалять, изменять и т. д.

Метод `Файлы.Найти()` как раз выполняет такой поиск. Если каталог, в котором вы ищете, не существует, «1С:Элемент» выбросит исключение, которое вы можете перехватить. Например:

```
метод Скрипт()
    попытка
        пер СписокФайлов = Файлы.Найти("D:\\Источник")
    поймать НетПути: ИсключениеФайловойСистемы
        Консоль.Записать("Нет каталога-источника. " + НетПути.Описание)
    ;
;
```

Если каталог **D:\Источник** отсутствует, в терминал будет выведено следующее сообщение:

```
Нет каталога-источника. Файл с именем 'Источник1' не существует
```

Скопировать

Скопировать выбранные файлы

Порой нужно скопировать все HTML-страницы из каталога `Источник` в каталог `Цель`. Чтобы найти все такие файлы, используйте `НастройкиПоискаФайлов` и регулярное выражение *(187)* для поиска файлов.

```
метод Скрипт()
    знч ПоискПоРегулярномуВыражению = новый НастройкиПоискаФайлов().ИмяСоответствует('.*\.html?')
    для Файл из Файлы.Найти("D:\\Источник", ПоискПоРегулярномуВыражению)
        Файлы.Скопировать(Файл.Путь, "D:\\Цель\\" + Файл.Имя)
    ;
;
```

В результате из каталога **D:\Источник** в каталог **D:\Цель** будут скопированы все файлы с расширениями **HTM**, **htm**, **HTML** или **html**.

Обратите внимание: тут вы в первый раз сталкиваетесь с *текучим интерфейсом*. Суть текучего интерфейса (Fluent API) заключается в том, что методы возвращают контекст своего вызова, благодаря чему упрощается множественный вызов методов одного экземпляра. Внешне это выглядит как цепочка методов, вызываемых последовательно.

В этом примере ещё нет такой цепочки, но конструктор **новый НастройкиПоискаФайлов()** возвращает экземпляр, от которого выполняется метод **ИмяСоответствует('.*\.html?')**. Так вот, этот метод возвращает тот же самый экземпляр. В результате он оказывается в переменной **ПоискПоРегулярномуВыражению**.

Скопировать всё с каталогами

```
метод Скрипт()
    Файлы.Скопировать("D:\\Источник", "D:\\Цель")
;
```

В результате всё содержимое каталога **D:\Источник**, вместе с вложенными каталогами и их файлами, будет скопировано в каталог **D:\Цель**. В каталоге **D:\Цель** будут созданы отсутствующие каталоги. Также будет создан и сам каталог **D:\Цель**, если он отсутствует.

Скопировать только файлы первого уровня

Для этого используйте `НастройкиПоискаФайлов` и его свойства, позволяющие ограничить область поиска.

```
метод Скрипт()
    знч НастройкиПоиска = новый НастройкиПоискаФайлов().МаксимальнаяГлубина(1).ИсключитьКаталоги(Истина)
    для Файл из Файлы.Найти("D:\\Источник", НастройкиПоиска)
        Файлы.Скопировать(Файл.Путь, "D:\\Цель\\" + Файл.Имя)
    ;
;
```

Обратите внимание, здесь вы уже видите цепочку вызовов в стиле текучего интерфейса: **.МаксимальнаяГлубина(1).ИсключитьКаталоги(Истина)**. Это возможно потому, что каждый из этих методов возвращает контекст своего вызова. Если бы этого не было, пришлось бы писать следующий код:

```
знч НастройкиПоиска = новый НастройкиПоискаФайлов()
НастройкиПоиска.МаксимальнаяГлубина(1)
НастройкиПоиска.ИсключитьКаталоги(Истина)
```

Перезаписать файл

```
метод Скрипт()
    Файлы.Скопировать("D:\\Источник\\документ.docx", "D:\\Цель\\документ.docx")
;
```

Не перезаписывать существующие файлы

Чтобы копировать файлы только в том случае, когда в целевом каталоге их ещё нет, используйте `НастройкиКопированияФайлов`.

```
метод Скрипт()
    знч НастройкиКопирования = новый НастройкиКопированияФайлов().ПропускатьСуществующие(Истина)
    Файлы.Скопировать("D:\\Источник\\документ.docx", "D:\\Цель\\документ.docx", НастройкиКопирования)
;
```

Переместить файл

Перемещение файлов работает так же, как и копирование, с той разницей, что файл-источник удаляется.

```
метод Скрипт()
    Файлы.Переместить("D:\\Источник\\документ.docx", "D:\\Цель\\документ.docx")
;
```

Чтобы переместить несколько файлов, сначала выберите нужные файлы, а затем переместите их. Примеры выбора файлов есть в разделе Скопировать *(246)*.

Удалить файл

Для удаления одного файла выполните следующий код:

```
метод  Скрипт()
      Файлы.Удалить("D:\\Цель\\Запрос.txt")
;
```

Если окажется, что этого файла уже нет (например, вы удалили его в другом месте алгоритма), «1С:Элемент» выкинет **ИсключениеФайловойСистемы** с описанием **Файл с именем 'Запрос.txt' не существует**.

Чтобы этого не происходило, передайте вторым параметром значение `Истина`. Тогда удаление будет выполняться в «тихом» режиме без выбрасывания исключений.

```
метод  Скрипт()
      Файлы.Удалить("D:\\Цель\\Запрос.txt",  Истина)
;
```

Чтобы очистить каталог от файлов, сначала выберите нужные файлы, а затем удалите их. Примеры выбора файлов есть в разделе Скопировать *(246)*.

Создать временный каталог или временный файл

Порой для записи логов или журналов работы вашего скрипта приходится создавать временные файлы. Также для различных преобразований вам может понадобиться временный каталог для записи туда служебных данных, которые нужны только на время работы вашего скрипта.

Скрипты могут работать на разных операционных системах. Описывать для каждой из них, в каком именно месте должны размещаться эти временные файлы и каталоги, — не очень хорошее занятие. Проще и удобнее воспользоваться возможностями, которые предоставляет «1С:Элемент».

Создать временный каталог

```
метод  Скрипт()
      знч  Временный  =  Файлы.СоздатьВременныйКаталог()
;
```

В результате будет создан, например, каталог **C:\Users*имя-пользователя*\AppData\Local\Temp\9989499669846057547**.

Каталог создаётся в каталоге временных файлов пользователя. Имя каталога генерируется автоматически.

В операционной системе Windows на каталог временных файлов пользователя указывает системная переменная **%TEMP%** (можно ввести это обозначение прямо в Проводник). Для ОС Windows 10 Pro это будет **C:\Users*имя-пользователя*\AppData\Local\Temp**.

Чтобы этот каталог автоматически удалялся после завершения работы вашего скрипта, вторым параметром передайте значение `Истина`. В первом параметре можно использовать собственный префикс, чтобы вы могли легче найти этот каталог визуально в Проводнике.

```
метод  Скрипт()
      знч  Временный  =  Файлы.СоздатьВременныйКаталог("MyScript",  Истина)
;
```

В результате на время работы скрипта будет создан, например, каталог **C:\Users*имя-пользователя*\AppData\Local\Temp\MyScript17552597362534176049**.

Создать временный файл

Если вам не нужен целый каталог, а достаточно только одного временного файла, его можно создать так:

```
метод Скрипт()
    знч Временный = Файлы.СоздатьВременныйФайл()
;
```

В результате будет создан, например, файл **C:\Users*имя-пользователя*\AppData\Local\Temp\9689108622517921372.tmp**.

Файл, так же как и каталог, создаётся в каталоге временных файлов пользователя, на который в ОС Windows указывает системная переменная **%TEMP%**.

Для файла вы тоже можете указать, чтобы он автоматически удалялся после завершения работы вашего скрипта, можете указать префикс и суффикс, чтобы вы могли легче найти этот каталог визуально в Проводнике.

```
метод Скрипт()
    знч Временный = Файлы.СоздатьВременныйФайл("MyScript", ".log", Истина)
;
```

В результате, на время работы скрипта будет создан, например, файл **C:\Users*имя-пользователя*\AppData\Local\Temp\MyScript495437552108960996.log**.

Домашний каталог пользователя

Ещё одним каталогом, который может понадобиться вам при работе, является домашний каталог пользователя. В ОС Windows на него указывает системная переменная **%USERPROFILE%**, а находится он, в ОС Windows 10 Pro, например, тут: **C:\Users*имя-пользователя***.

В этом каталоге хранятся настройки приложений, привязанные к пользователю и к компьютеру. Получить его можно так:

```
метод Скрипт()
    знч КаталогПользователя = Файлы.ПолучитьДомашнийКаталогПользователя()
;
```

Отследить появление файла в каталоге

Один из способов взаимодействия программных систем друг с другом — это файловый обмен. Например, у системы **А** есть каталог, через который осуществляется взаимодействие.

Если система **Б** хочет получить какие-то данные, она помещает в этот каталог *сигнальный файл*. Сигнальный файл — это пустой файл со специальным именем. Наличие сигнального файла указывает на то, что требуется выполнить какие-то действия или что данные готовы к отправке. Допустим, сигнальный файл называется **Запрос.txt**.

Система **А** отслеживает появление новых файлов в этом каталоге. Она видит, что появился сигнальный файл **Запрос.txt**. Это говорит ей о том, что нужно подготовить данные и записать их в файл. Она готовит и записывает данные в файл **Данные.txt**. Данных может быть много, файл может быть большим, его запись может занять некоторое время. Когда запись файла **Данные.txt** закончена, система удаляет сигнальный файл **Запрос.txt** и записывает другой сигнальный файл — **Ответ.txt**.

Система **Б** тоже отслеживает появление новых файлов в этом каталоге. Она видит, что появился сигнальный файл **Ответ.txt**. Для неё это является сигналом о том, что нужно забрать файл **Данные.txt**. Она забирает его и удаляет файл **Ответ.txt**. Операция закончена.

Главным моментом в таком взаимодействии является возможность системы **А** и системы **Б** отслеживать изменение файлов в каталоге. В данном случае — отслеживать появление там новых сигнальных файлов.

Для того чтобы организовать аналогичное отслеживание каталога в «1С:Элементе», используйте `МониторФайловойСистемы` и обрабатывайте события, возвращаемые методом `МониторФайловойСистемы.ПолучитьСобытия()`.

```
метод Скрипт()
    исп Монитор = новый МониторФайловойСистемы()
    Монитор.Отслеживать(новый Файл("D:\\Цель\\Новая папка"))
    пока Истина
        пер МассивСобытий = Монитор.ПолучитьСобытия()
        для Событие из МассивСобытий
            если Событие.ВидСобытия == ВидСобытияФайловойСистемы.Создание
                если Событие.Файл.Имя == "Запрос.txt"
                    Консоль.Записать("Пришёл запрос на получение данных")
                ;
            ;
        ;
    ;
;
```

В результате, как только вы поместите в каталог **D:\Цель** файл **Запрос.txt**, в терминал будет выведено сообщение **Пришёл запрос на получение данных**.

> **Примечание:**
>
> Подробнее о работе с файлами читайте в руководстве разработчика и в справке по стандартной библиотеке.

Чтение и запись текстовых файлов

Файл — Поток — Чтение

Каждый раз, когда вы будете работать с какими-либо файлами, вы будете проходить через последовательность трёх типов:

- `Файл`,
- какой-либо поток,
- специализированный тип для работы именно с этим видом файлов.

Это касается не только текстовых файлов, но и любых других: .json-файлов, .xml-файлов, .zip-файлов и т. д.

Тип `Файл` описывает файл и больше не выполняет практически никаких действий. Он может предоставить разную информацию о файле (размер, момент последнего изменения и т. д.), но основное его назначение в другом. Экземпляр этого типа, грубо говоря, указывает на то место на диске, из которого нужно взять данные. Либо на то место, куда их нужно положить (записать).

Следующие типы в этой цепочке — это `ПотокЧтения` и `ПотокЗаписи`. Они представляют собой чисто технический, транспортный уровень работы с файлами. Они ничего не знают о специфике файла. Они лишь могут последовательно прочитать информацию, находящуюся в файле, или последовательно её туда записать. Что именно это за информация, из чего она состоит, можно ли как-то работать с разными частями этой информации — они не знают.

И, наконец, типы `ЧтениеДанных` и `ЗаписьДанных`. Вот эти типы как раз предназначены для работы с текстовыми файлами. Они, например, знают, что текстовые файлы состоят из строк. Поэтому они умеют читать их по строкам — `ЧтениеДанных.ПрочитатьСтроку()` — и так же записывать — `ЗаписьДанных.ЗаписатьСтроку()`. Они знают, что файлы могут иметь разные кодировки символов, разные разделители строк и т. д. и позволяют настраивать эти параметры при чтении и записи.

Есть у них и другие возможности, которые используются не часто, но могут быть полезны — например, чтение и запись целых чисел, чтение и запись нескольких символов.

Вы не будете вникать во все подробности, а познакомитесь только с двумя простыми возможностями чтения и записи текстовых файлов, которые используются наиболее часто.

Главное, что нужно понять из этого вступления: существует последовательность типов, экземпляры которых взаимодействуют друг с другом. Когда вы читаете, это `Файл` — `ПотокЧтения` и `ЧтениеДанных`. Когда вы записываете, происходит всё то же самое, только в обратную сторону: `ЗаписьДанных` — `ПотокЗаписи` — `Файл`.

Чтение файла целиком

Пусть существует файл **D:\\стих.txt**, который содержит следующий текст:

```
У лукоморья дуб зелёный;
Златая цепь на дубе том:
И днём и ночью кот учёный
Всё ходит по цепи кругом;
Идёт направо - песнь заводит,
Налево - сказку говорит.
Там чудеса: там леший бродит,
Русалка на ветвях сидит;
```

Чтобы прочитать этот файл, создайте экземпляр `Файл` и на его основе создайте экземпляр `ПотокЧтения`. Поскольку вам нужен весь файл целиком, вы можете получить его сразу из потока методом `ПотокЧтения.ПрочитатьКакСтроку()`.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\стих.txt").ОткрытьПотокЧтения()
```

```
    пер Стих = ПотокЧтения.ПрочитатьКакСтроку()
;
```

В результате в переменной **Стих** будет следующее значение:

```
У лукоморья дуб зелёный;
Златая цепь на дубе том:
И днём и ночью кот учёный
Всё ходит по цепи кругом;
Идёт направо - песнь заводит,
Налево - сказку говорит.
Там чудеса: там леший бродит,
Русалка на ветвях сидит;
```

Запись файла целиком

Чтобы записать файл целиком, нужно создать экземпляр `Файл` и на его основе создать экземпляр `ПотокЗаписи`. После этого можно сразу записать весь файл, передав его содержимое в поток методом `ПотокЗаписи.Записать()`.

```
метод Скрипт()
    пер Текст =
        "У лукоморья дуб зелёный;
        Златая цепь на дубе том:
        И днём и ночью кот учёный
        Всё ходит по цепи кругом;
        Идёт направо - песнь заводит,
        Налево - сказку говорит.
        Там чудеса: там леший бродит,
        Русалка на ветвях сидит;"
    исп ПотокЗаписи = новый Файл("D:\\стих.txt").ОткрытьПотокЗаписи()
    пер Стих = ПотокЗаписи.Записать(Текст)
;
```

В результате на диске **D:** появится файл **стих.txt** с содержимым, которое показано в листинге.

Чтение файла по строкам

Чтобы прочитать файл по отдельным строкам, используйте тип `ЧтениеДанных`.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\стих.txt").ОткрытьПотокЧтения()
    знч ЧтениеДанных = новый ЧтениеДанных(ПотокЧтения)
    пока не ЧтениеДанных.ЧтениеЗавершено()
        Консоль.Записать(ЧтениеДанных.ПрочитатьСтроку())
    ;
;
```

В результате в терминал будут выведены следующие сообщения.

```
У лукоморья дуб зелёный;
Златая цепь на дубе том:
И днём и ночью кот учёный
Всё ходит по цепи кругом;
```

```
Идёт направо - песнь заводит,
Налево - сказку говорит.
Там чудеса: там леший бродит,
Русалка на ветвях сидит;
```

Запись файла по строкам

Чтобы записать файл по отдельным строкам, используйте тип `ЗаписьДанных`.

```
метод Скрипт()
	исп ПотокЗаписи = новый Файл("D:\\стих.txt").ОткрытьПотокЗаписи()
	знч ЗаписьДанных = новый ЗаписьДанных(ПотокЗаписи)
	для Строка из ПолучитьСтроки()
		ЗаписьДанных.ЗаписатьСтроку(Строка)
	;
;

метод ПолучитьСтроки(): Массив<Строка>
	пер СтихотворныеСтроки = новый Массив<Строка>()
	СтихотворныеСтроки.Добавить("У лукоморья дуб зелёный;")
	СтихотворныеСтроки.Добавить("Златая цепь на дубе том:")
	СтихотворныеСтроки.Добавить("И днём и ночью кот учёный")
	СтихотворныеСтроки.Добавить("Всё ходит по цепи кругом;")
	СтихотворныеСтроки.Добавить("Идёт направо - песнь заводит,")
	СтихотворныеСтроки.Добавить("Налево - сказку говорит.")
	СтихотворныеСтроки.Добавить("Там чудеса: там леший бродит,")
	СтихотворныеСтроки.Добавить("Русалка на ветвях сидит;")
	возврат СтихотворныеСтроки
;
```

В результате на диске **D:** появится файл **стих.txt** с содержимым, которое показано в листинге.

Указание кодировки при записи и чтении

Порой случается, что вы прочитали текстовый файл, а ваш результат выглядит, например, так:

```
□ □□□□□□□□□□ □□□ □□□□□□;
□□□□□□□ □□□□ □□ □□□□ □□□:
□ □□□ □ □□□□□ □□□ □□□□□□□
□□ □□□□□ □□ □□□□ □□□□□□□;
□□□ □□□□□□□□ - □□□□□ □□□□□□□□,
□□□□□□□ - □□□□□□□ □□□□□□□□.
□□□ □□□□□□□: □□□ □□□□□ □□□□□□□,
□□□□□□□□ □□ □□□□□□□ □□□□□;
```

Это верный признак того, что вы прочитали файл в неподходящей кодировке. «1С:Элемент», если вы не предпринимаете специальных действий, читает файлы в кодировке UTF-8.

Для указания кодировки создайте экземпляр `ЧтениеДанных`, установите кодировку и передайте этот экземпляр в конструктор `ЧтениеДанных`.

```
метод Скрипт()
	исп ПотокЧтения = новый Файл("D:\\стих.txt").ОткрытьПотокЧтения()
	знч Настройки = новый НастройкиЧтенияДанных()
	Настройки.Кодировка = "ansi-1251"
```

```
    знч ЧтениеДанных = новый ЧтениеДанных(ПотокЧтения, Настройки)
    пер Стих = ЧтениеДанных.ПрочитатьСимволы()
;
```

Кодировка задаётся строкой, доступные варианты кодировок смотрите в руководстве разработчика, в таблице псевдонимов.

Безопасное чтение и запись файлов

Бывает так, что при чтении файлов вы ошиблись в имени. Или, наоборот, вы не ошиблись, но нужного файла не оказалось на диске. Чтобы обрабатывать такие исключения, читайте и пишите файлы в инструкции `попытка` и перехватывайте `ИсключениеВводаВывода`. Например, так:

```
метод Скрипт()
    попытка
        исп ПотокЧтения = новый Файл("D:\\стих.txt").ОткрытьПотокЧтения()
        знч ЧтениеДанных = новый ЧтениеДанных(ПотокЧтения)
        пер Стих = ЧтениеДанных.ПрочитатьСимволы()
    поймать Искл: ИсключениеВводаВывода
        Консоль.Записать(Искл.Описание)
    ;
;
```

JSON

Общая информация

В настоящее время в веб-приложениях широко используется формат обмена данными JSON. JSON (JavaScript Object Notation) — это текстовый формат обмена данными, с которым могут работать все браузеры. Этот формат похож на XML, но по сравнению с XML он является более лаконичным и требует меньше места.

Существует несколько основных сценариев использования JSON:

- Интеграция с внешними системами через их HTTP-интерфейсы: Google Calendar, Salesforce.com, REST-интерфейс «1С:Предприятия», SharePoint и т. д.
- Обмен .json-файлами с внешними системами. Формирование конфигурационных, настроечных файлов. Использование их в процедурах обмена данными, например с интернет-магазинами.
- Использование .json-файлов для обмена данными между разными скриптами.

JSON — это текстовый формат, поэтому данные в формате JSON могут содержать:

- *Объект* — неупорядоченное множество пар ***ключ:значение***, заключённых в фигурные скобки «{» и «}». Пары ***ключ:значение*** разделяются запятыми «,». Например:

```
{
    "Код" : "000000006",
    "Наименование" : "АО Пантера",
    "ОбъемПродаж" : 2500000,
}
```

- *Массив* — множество значений. Массив заключается в квадратные скобки «[» и «]». Значения разделяются запятыми «,». Например:

```
[
    "+7(999)890-987",
    "+7(999)261-79-79",
    "+7(999)789-67-85"
]
```

- *Значение* — может быть строкой, числом, объектом, массивом или литералом `true`, `false`, `null`.
  - `Строка` — набор символов, заключённый в двойные кавычки «"» и «"».
  - `Число` — сериализуется с разделителем точка «.». Точность числа не ограничена.

.json-файл всегда содержит один корневой элемент, в который вложены остальные. Корневым элементом может быть либо объект, либо массив. Пример .json-файла:

```
{
    "Код" : "000000017",
    "Наименование" : "ОАО Топаз",
    "Телефоны" : [
                    "+7(999)528-96-21",
                    "+7(999)456-87-68"
                ],
    "ОбъемПродаж" : 5000000,
    "ЭтоПоставщик" : false
}
```

В этом примере корневым элементом является объект, он ограничен фигурными скобками «{» и «}».

Объект содержит пары **ключ:значение** с ключами: **Код**, **Наименование**, **Телефоны**, **ОбъемПродаж** и **ЭтоПоставщик**. Все ключи имеют простые значения, кроме ключа **Телефоны**. Его значением является массив телефонных номеров. Массив ограничен квадратными скобками «[» и «]».

Таким образом, в формате JSON с помощью перечисленных элементов можно описать сущности любой сложности, вкладывая их друг в друга. В «1С:Элементе» реализовано несколько слоёв работы с JSON.

Самые универсальные и гибкие — это низкоуровневые средства *потоковой записи и чтения*. Более высокоуровневые и не такие универсальные — средства сериализации коллекций в JSON и средства их десериализации.

Потоковая запись

Для потоковой записи .json-файла используйте тип `ЗаписьJson`. Он последовательно, от значения к значению, записывает JSON в файл. Таким образом, запись JSON происходит без формирования всего документа в памяти. Это является существенным преимуществом, когда объем .json-документа может быть большим или когда он заранее неизвестен. То же самое касается и чтения .json-файлов.

Вот пример записи массива партнёров. Экземпляр `ЗаписьJson` конструируется из потока двоичных данных, в который будет выполняться запись.

```
знч Запись = новый ЗаписьJson(ПотокЗаписи)
```

Про каждого партнёра записывается следующая информация: **Код**, **Наименование**, **Телефоны** (массив строк), **ОбъемПродаж** и булево свойство **ЭтоПоставщик**. Содержимое скрипта поясняется в комментариях.

Здесь вы в полной мере видите использование текучего интерфейса, о котором говорилось в разделе Работа с файлами *(246)*.

```
метод Скрипт()
    исп ПотокЗаписи = новый Файл("D:\\пример.json").ОткрытьПотокЗаписи()

    знч Запись = новый ЗаписьJson(ПотокЗаписи)
    Запись.ЗаписатьНачалоМассива()                    // начало корневого элемента - массива
        .ЗаписатьНачалоОбъекта()                      // партнёр Топаз - первый элемент массива, объект
            .Записать("Код", "000000017")             // Код
            .Записать("Наименование", "ОАО Топаз")    // Наименование
            .ЗаписатьИмяСвойства("Телефоны")          // Телефоны - имя свойства
            .ЗаписатьНачалоМассива()                  // значения телефонов - массив
                .Записать("+7(999)528-96-21")
                .Записать("+7(999)456-87-68")
            .ЗаписатьКонецМассива()                   // конец массива телефонов
            .Записать("ОбъемПродаж", 5000000)         // ОбъемПродаж
            .Записать("ЭтоПоставщик", Ложь)           // ЭтоПоставщик
        .ЗаписатьКонецОбъекта()                       // конец объекта Топаз
        .ЗаписатьНачалоОбъекта()                      // партнёр Пантера - второй элемент массива, объект
            .Записать("Код", "000000006")
            .Записать("Наименование", "АО Пантера")
            .ЗаписатьИмяСвойства("Телефоны")
            .ЗаписатьНачалоМассива()
                .Записать("+7(999)890-987").Записать("+7(999)261-79-79").Записать("+7(999)789-67-85")
            .ЗаписатьКонецМассива()
            .Записать("ОбъемПродаж", 2500000)
            .Записать("ЭтоПоставщик", Истина)
        .ЗаписатьКонецОбъекта()
    .ЗаписатьКонецМассива()                           // конец корневого объекта - массива
;
```

В результате на диск **D:** будет записан файл **пример.json** со следующим содержимым:

```
[
  {
    "Код" : "000000017",
    "Наименование" : "ОАО Топаз",
    "Телефоны" : [
      "+7(999)528-96-21",
      "+7(999)456-87-68"
    ],
    "ОбъемПродаж" : 5000000,
    "ЭтоПоставщик" : false
  },
  {
    "Код" : "000000006",
    "Наименование" : "АО Пантера",
    "Телефоны" : [
      "+7(999)890-987",
      "+7(999)261-79-79",
      "+7(999)789-67-85"
    ],
```

```
    "ОбъемПродаж" : 2500000,
    "ЭтоПоставщик" : true
  }
]
```

Потоковое чтение

Для потокового чтения .json-документа используйте тип `ЧтениеJson`. Экземпляр `ЧтениеJson` конструируется из потока двоичных данных, из которого будет выполняться чтение.

```
знч Чтение = новый ЧтениеJson(ПотокЧтения)
```

Файл **пример.json**, полученный в предыдущем примере, можно прочитать следующим образом.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\пример.json").ОткрытьПотокЧтения()

    знч Чтение = новый ЧтениеJson(ПотокЧтения)
    пока Чтение.Следующий()
        выбор Чтение.ВидУзла
        когда ИмяСвойства
            Консоль.Записать("Имя " + Чтение.Значение)
        когда Булево, Строка, Число, Комментарий
            Консоль.Записать("Значение " + Чтение.Значение)
        ;
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Имя Код
Значение 000000017
Имя Наименование
Значение ОАО Топаз
Имя Телефоны
Значение +7(999)528-96-21
Значение +7(999)456-87-68
Имя ОбъемПродаж
Значение 5000000
Имя ЭтоПоставщик
Значение false
Имя Код
Значение 000000006
Имя Наименование
Значение АО Пантера
Имя Телефоны
Значение +7(999)890-987
Значение +7(999)261-79-79
Значение +7(999)789-67-85
Имя ОбъемПродаж
Значение 2500000
Имя ЭтоПоставщик
Значение true
```

Работа со строкой JSON

Данные в формате JSON нужно получать и читать не только в виде файла, но и в виде строки. Это может понадобиться при работе с HTTP-сервисами, так как тело HTTP-запроса представляет собой строку JSON.

В таком случае можно записывать JSON-данные не в файл, а в строку и затем подставлять эту строку в тело HTTP-запроса. И, наоборот, можно брать строку, которая является телом HTTP-запроса, и читать из неё JSON-данные с помощью объекта `ЧтениеJson`.

Запись JSON в строку

Чтобы записать данные JSON в строку, нужно конструировать экземпляр `ЗаписьJson` из строкового потока.

```
исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
знч Запись = новый ЗаписьJson(ПотокЗаписи)
```

После того как все элементы JSON записаны, итоговую строку JSON можно получить методом `СтроковыйПотокЗаписи.ВСтроку()`.

```
пер JsonСтрока = ПотокЗаписи.ВСтроку()
```

Например:

```
метод Скрипт()
    исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
    знч Запись = новый ЗаписьJson(ПотокЗаписи)
    СформироватьJson(Запись)
    пер JsonСтрока = ПотокЗаписи.ВСтроку()
;

метод СформироватьJson(Запись: ЗаписьJson)
    Запись.ЗаписатьНачалоОбъекта()
        .Записать("Наименование", "ОАО Топаз")
        .ЗаписатьИмяСвойства("Телефоны")
        .ЗаписатьНачалоМассива()
            .Записать("+7(999)528-96-21")
            .Записать("+7(999)456-87-68")
        .ЗаписатьКонецМассива()
        .Записать("ОбъемПродаж", 5000000)
        .Записать("ЭтоПоставщик", Ложь)
    .ЗаписатьКонецОбъекта()
;
```

В результате в переменной **JsonСтрока** будет следующее значение:

```
{
  "Наименование" : "ОАО Топаз",
  "Телефоны" : [
    "+7(999)528-96-21",
    "+7(999)456-87-68"
  ],
  "ОбъемПродаж" : 5000000,
  "ЭтоПоставщик" : false
}
```

Чтение JSON из строки

Чтобы прочитать данные JSON из строки, экземпляр `ЧтениеJson` нужно конструировать из этой самой строки.

```
пер JsonСтрока = СформироватьJson()
знч Чтение = новый ЧтениеJson(JsonСтрока)
```

Например:

```
метод Скрипт()
    пер JsonСтрока = СформироватьJson()
    знч Чтение = новый ЧтениеJson(JsonСтрока)
    пока Чтение.Следующий()
        выбор Чтение.ВидУзла
        когда ИмяСвойства
            Консоль.Записать("Имя " + Чтение.Значение)
        когда Булево, Строка, Число, Комментарий
            Консоль.Записать("Значение " + Чтение.Значение)
        ;
    ;
;

метод СформироватьJson():Строка
    пер JsonСтрока =
        "{
            \"Наименование\" : \"ОАО Топаз\",
            \"Телефоны\" : [
                \"+7(999)528-96-21\",
                \"+7(999)456-87-68\"
            ],
            \"ОбъемПродаж\" : 5000000,
            \"ЭтоПоставщик\" : false
        }"
    возврат JsonСтрока
;
```

В результате в терминал будут выведены следующие сообщения:

```
Имя Наименование
Значение ОАО Топаз
Имя Телефоны
Значение +7(999)528-96-21
Значение +7(999)456-87-68
Имя ОбъемПродаж
Значение 5000000
Имя ЭтоПоставщик
Значение false
```

Проверка структуры записываемого документа

В то время как вы формируете JSON, `ЗаписьJson` сразу же проверяет его структуру. Если вы решите записать неподходящий в данном месте элемент JSON, «1С:Элемент» сообщит вам об этом, выкинув `ИсключениеЗаписиJson`.

При записи очень больших документов JSON вы, возможно, захотите отключить эту проверку, чтобы ускорить запись. Это можно сделать.

Для этого нужно сконструировать экземпляр настроек, установить свойство `НастройкиЗаписиJson.ПроверятьСтруктуру` в значение `Ложь` и передать эти настройки в конструктор `ЗаписьJson`.

```
знч Настройки = новый НастройкиЗаписиJson()
Настройки.ПроверятьСтруктуру = Ложь
знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
```

Например:

```
метод Скрипт()
	исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
	знч Настройки = новый НастройкиЗаписиJson()
	Настройки.ПроверятьСтруктуру = Ложь
	знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
	СформироватьJson(Запись)
	пер JsonСтрока = ПотокЗаписи.ВСтроку()
;

метод СформироватьJson(Запись: ЗаписьJson)
	Запись.ЗаписатьНачалоОбъекта()
		.Записать("Наименование", "ОАО Топаз")
		.ЗаписатьИмяСвойства("Телефоны")
		.ЗаписатьНачалоМассива()
			.Записать("+7(999)528-96-21")
			.Записать("+7(999)456-87-68")
		.ЗаписатьКонецМассива()
		.Записать("ОбъемПродаж", 5000000)
		.Записать("ЭтоПоставщик", Ложь)
	.ЗаписатьКонецОбъекта()
;
```

Управление переносом строк

Если не предпринимать специальных действий, то `ЗаписьJson` записывает JSON в несколько строк. Это удобно для визуального контроля.

Однако некоторые HTTP-сервисы требуют, чтобы JSON был записан в одну строку. Это можно сделать.

Для этого нужно сконструировать экземпляр настроек, установить свойство `НастройкиЗаписиJson.ПереносСтрок` в значение `ПереносСтрокJson.Отсутствует` и передать эти настройки в конструктор `ЗаписьJson`.

```
знч Настройки = новый НастройкиЗаписиJson()
Настройки.ПереносСтрок = ПереносСтрокJson.Отсутствует
знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
```

Например:

```
метод Скрипт()
    исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
    знч Настройки = новый НастройкиЗаписиJson()
    Настройки.ПереносСтрок = ПереносСтрокJson.Отсутствует
    знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
    СформироватьJson(Запись)
    пер JsonСтрока = ПотокЗаписи.ВСтроку()
;

метод СформироватьJson(Запись: ЗаписьJson)
    Запись.ЗаписатьНачалоОбъекта()
        .Записать("Наименование", "ОАО Топаз")
        .ЗаписатьИмяСвойства("Телефоны")
        .ЗаписатьНачалоМассива()
            .Записать("+7(999)528-96-21")
            .Записать("+7(999)456-87-68")
        .ЗаписатьКонецМассива()
        .Записать("ОбъемПродаж", 5000000)
        .Записать("ЭтоПоставщик", Ложь)
    .ЗаписатьКонецОбъекта()
;
```

В результате в переменной **JsonСтрока** будет следующее значение:

```
{"Наименование":"ОАО Топаз","Телефоны":["+7(999)528-96-21","+7(999)456-87-68"],"ОбъемПродаж":5000000,"ЭтоПоставщик":false}
```

Управление синтаксическим отступом

По умолчанию `ЗаписьJson` добавляет к строкам JSON синтаксический отступ, состоящий из двух пробелов. Возможно, вы захотите увеличить его, чтобы текст JSON лучше читался. Это можно сделать.

Для этого нужно сконструировать экземпляр настроек, установить свойство `НастройкиЗаписиJson.СимволыОтступа` в значение `Символы.ТАБ`, например, и передать эти настройки в конструктор `ЗаписьJson`.

```
знч Настройки = новый НастройкиЗаписиJson()
Настройки.СимволыОтступа = Символы.ТАБ
знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
```

Например:

```
метод Скрипт()
    исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
    знч Настройки = новый НастройкиЗаписиJson()
    Настройки.СимволыОтступа = Символы.ТАБ
    знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
    СформироватьJson(Запись)
    пер JsonСтрока = ПотокЗаписи.ВСтроку()
;

метод СформироватьJson(Запись: ЗаписьJson)
    Запись.ЗаписатьНачалоОбъекта()
        .Записать("Наименование", "ОАО Топаз")
        .ЗаписатьИмяСвойства("Телефоны")
```

```
            .ЗаписатьНачалоМассива()
                .Записать("+7(999)528-96-21")
                .Записать("+7(999)456-87-68")
            .ЗаписатьКонецМассива()
            .Записать("ОбъемПродаж", 5000000)
            .Записать("ЭтоПоставщик", Ложь)
        .ЗаписатьКонецОбъекта()
;
```

В результате в переменной **JsonСтрока** будет следующее значение:

```
{
 "Наименование" : "ОАО Топаз",
 "Телефоны" : [
  "+7(999)528-96-21",
  "+7(999)456-87-68"
 ],
 "ОбъемПродаж" : 5000000,
 "ЭтоПоставщик" : false
}
```

Чтобы, наоборот, избавиться от синтаксических отступов, установите свойство `НастройкиЗаписиJson.СимволыОтступа` в значение `""` (пустая строка). Результат будет следующим:

```
{
"Наименование" : "ОАО Топаз",
"Телефоны" : [
"+7(999)528-96-21",
"+7(999)456-87-68"
],
"ОбъемПродаж" : 5000000,
"ЭтоПоставщик" : false
}
```

Экранирование символов

Внешние системы могут использовать разные нотации JSON, могут считать те или иные символы недопустимыми, и так далее. Чтобы внешняя система восприняла тот или иной символ не как управляющий, а как обычный символ, который нужно записать, этот символ экранируют, то есть помечают специальным образом.

Для этого нужно сконструировать экземпляр настроек `НастройкиЗаписиJson` и установить одно из его свойств, которое вам нужно:

- `ЭкранированиеСимволов`;
- `ЭкранироватьАмперсанд`;
- `ЭкранироватьОдинарныеКавычки`;
- `ЭкранироватьРазделителиСтрокИАбзацев`;
- `ЭкранироватьСлеш`;
- `ЭкранироватьУгловыеСкобки`.

После этого настройки нужно передать в конструктор `ЗаписьJson`. Например, вы хотите экранировать косую черту, которая есть в путях к файлам:

```
знч Настройки = новый НастройкиЗаписиJson()
Настройки.СимволыОтступа = Символы.ТАБ
знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
```

Полный пример будет выглядеть так:

```
метод Скрипт()
	исп ПотокЗаписи = новый СтроковыйПотокЗаписи()
	знч Настройки = новый НастройкиЗаписиJson()
	Настройки.ЭкранироватьСлеш = Истина
	знч Запись = новый ЗаписьJson(ПотокЗаписи, Настройки)
	СформироватьJson(Запись)
	пер JsonСтрока = ПотокЗаписи.ВСтроку()
;

метод СформироватьJson(Запись: ЗаписьJson)
	Запись.ЗаписатьНачалоОбъекта()
		.Записать("Путь", "C:/файл.txt")
	.ЗаписатьКонецОбъекта()
;
```

В результате в переменной **JsonСтрока** будет следующее значение:

```
{
  "Путь" : "C:\u002Fфайл.txt"
}
```

Здесь \u002F — это обозначение косой черты в кодировке Unicode.

Объектная техника

Во многих случаях (например, при обмене информацией с внешними системами, чтении конфигурационных файлов в формате JSON и др.) проще и удобнее использовать не последовательное чтение и последовательную запись, а объектную технику работы с JSON. Эта техника позволяет избежать рутинной работы по чтению и записи каждого отдельного значения или свойства.

При чтении элементы JSON отображаются в фиксированный набор типов «1С:Элемента»: `Строка`, `Число`, `Булево`, `Неопределено`, `Массив` и `Соответствие`. Этот процесс называется *десериализацией*.

При записи можно сформировать в памяти массив или соответствие с нужными данными и быстро записать их в файл JSON. Этот процесс называется *сериализацией*.

Платой за простоту работы является расход памяти. При сериализации вам нужно сначала сформировать в оперативной памяти всю структуру с данными — и только после этого записывать её в JSON. Поскольку вы сами формируете эту структуру, вы можете следить за её размером. Если вы увидите, что оперативная память начинает заканчиваться, вы можете записать данные не в один большой, а в несколько .json-файлов поменьше.

При десериализации всё намного хуже. Вы не можете знать заранее размеры .json-файлов, которые вам понадобится читать. Вполне может случиться так, что вам попадётся какой-нибудь лог-файл за несколько лет работы. Попытавшись прочитать его в память весь целиком, вы просто израсходуете всю оперативную память, и на этом работа скрипта закончится с ошибкой.

Поэтому объектную технику можно смело использовать в тех случаях, когда вы точно знаете, что размеры .json-файлов не будут очень большими. В противном случае лучше использовать смешанную технику *(267)* работы, когда файл читается последовательно, но отдельные его части, небольшие массивы или объекты, десериализуются с использованием объектной техники. То же самое касается и записи данных в JSON.

Десериализация

Если структура .json-файла известна вам заранее, вы можете весь его десериализовать в массив или соответствие.

Десериализация из файла

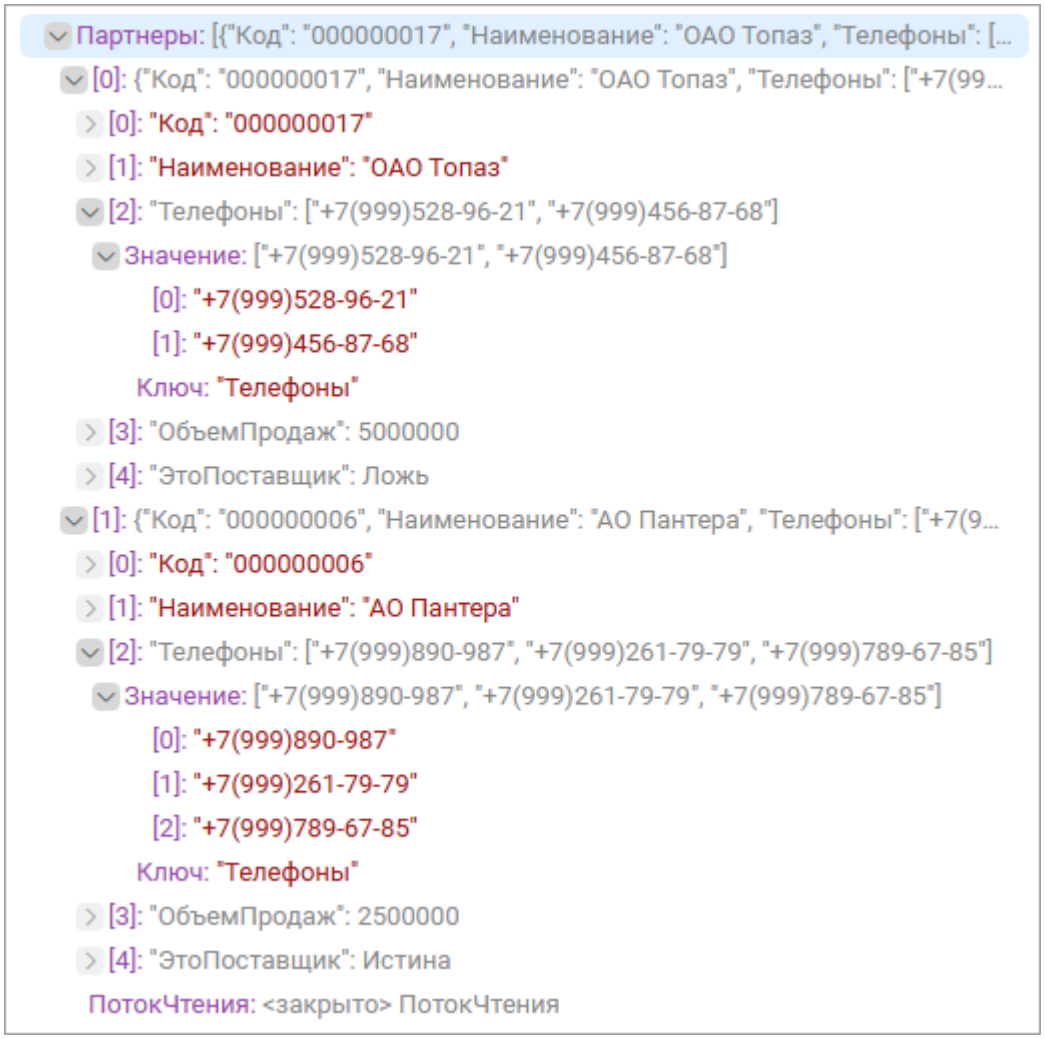
Например, файл **пример.json** содержит следующий текст.

```
[
  {
    "Код" : "000000017",
    "Наименование" : "ОАО Топаз",
    "Телефоны" : [
      "+7(999)528-96-21",
      "+7(999)456-87-68"
    ],
    "ОбъемПродаж" : 5000000,
    "ЭтоПоставщик" : false
  },
  {
    "Код" : "000000006",
    "Наименование" : "АО Пантера",
    "Телефоны" : [
      "+7(999)890-987",
      "+7(999)261-79-79",
      "+7(999)789-67-85"
    ],
    "ОбъемПродаж" : 2500000,
    "ЭтоПоставщик" : true
  }
]
```

С помощью метода `СериализацияJson.ПрочитатьМассив()` вы можете получить из него массив с данными.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\пример.json").ОткрытьПотокЧтения()
    пер Партнеры = СериализацияJson.ПрочитатьМассив(ПотокЧтения)
;
```

В результате в переменной **Партнеры** будет следующее значение:



Десериализация из строки

Пусть, для разнообразия, в строке JSON содержится объект.

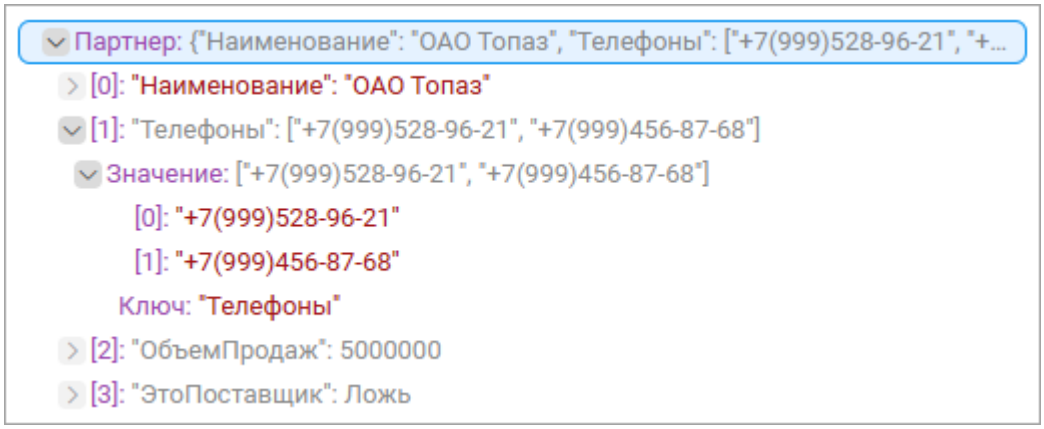
```
пер JsonСтрока =
        "{
            \"Наименование\" : \"ОАО Топаз\",
            \"Телефоны\" : [
                \"+7(999)528-96-21\",
                \"+7(999)456-87-68\"
            ],
            \"ОбъемПродаж\" : 5000000,
            \"ЭтоПоставщик\" : false
        }"
```

С помощью метода `СериализацияJson.ПрочитатьОбъект()` вы можете получить из него соответствие с данными. Чтобы десериализовать JSON из строки, нужно читать не поток двоичных данных, а саму строку, содержащую JSON.

```
метод Скрипт()
    пер JsonСтрока = СформироватьJson()
    пер Партнер = СериализацияJson.ПрочитатьОбъект(JsonСтрока)
;

метод СформироватьJson():Строка
```

```
    пер JsonСтрока =
        "{
            \"Наименование\" : \"ОАО Топаз\",
            \"Телефоны\" : [
                \"+7(999)528-96-21\",
                \"+7(999)456-87-68\"
            ],
            \"ОбъемПродаж\" : 5000000,
            \"ЭтоПоставщик\" : false
        }"
    возврат JsonСтрока
;
```

В результате в переменной **Партнер** будет следующее значение:



Сериализация

Сериализация в файл

Например, метод **ПолучитьСоответствие()** возвращает соответствие, которое нужно сериализовать в .json-файл. Используйте для этого метод `СериализацияJson.ЗаписатьОбъект()`.

```
метод Скрипт()
    исп ПотокЗаписи = новый Файл("D:\\пример.json").ОткрытьПотокЗаписи()
    пер Соответствие = ПолучитьСоответствие()
    СериализацияJson.ЗаписатьОбъект(ПотокЗаписи, Соответствие)
;

метод ПолучитьСоответствие():Объект?
    пер JsonСтрока =
        "{
            \"Наименование\" : \"ОАО Топаз\",
            \"Телефоны\" : [
                \"+7(999)528-96-21\",
                \"+7(999)456-87-68\"
            ],
            \"ОбъемПродаж\" : 5000000,
            \"ЭтоПоставщик\" : false
        }"
    возврат СериализацияJson.ПрочитатьОбъект(JsonСтрока)
;
```

В результате на диск **D:** будет записан файл **пример.json** со следующим содержимым:

```
{
  "Наименование" : "ОАО Топаз",
  "Телефоны" : [
    "+7(999)528-96-21",
    "+7(999)456-87-68"
  ],
  "ОбъемПродаж" : 5000000,
  "ЭтоПоставщик" : false
}
```

Сериализация в строку

Например, метод **ПолучитьСоответствие()** возвращает соответствие, которое нужно сериализовать в JSON-строку. Используйте для этого перегруженный метод `СериализацияJson.ЗаписатьОбъект()`, в который передаётся только сериализуемый объект.

```
метод Скрипт()
    исп ПотокЗаписи = новый Файл("D:\\пример.json").ОткрытьПотокЗаписи()
    пер Соответствие = ПолучитьСоответствие()
    пер JsonСтрока = СериализацияJson.ЗаписатьОбъект(Соответствие)
;

метод ПолучитьСоответствие():Объект?
    пер JsonСтрока =
        "{
            \"Наименование\" : \"ОАО Топаз\",
            \"Телефоны\" : [
                \"+7(999)528-96-21\",
                \"+7(999)456-87-68\"
            ],
            \"ОбъемПродаж\" : 5000000,
            \"ЭтоПоставщик\" : false
        }"
    возврат СериализацияJson.ПрочитатьОбъект(JsonСтрока)
;
```

В результате в переменной **JsonСтрока** будет следующее значение:

```
{
  "Наименование" : "ОАО Топаз",
  "Телефоны" : [
    "+7(999)528-96-21",
    "+7(999)456-87-68"
  ],
  "ОбъемПродаж" : 5000000,
  "ЭтоПоставщик" : false
}
```

Смешанная техника

Если вы знаете структуру .json-файла, то с помощью последовательной техники чтения можете дойти до некоторого JSON-элемента и десериализовать его в объектной технике. Передвигаясь таким образом по файлу, вы будете «кушать» его отдельными кусками, не расходуя большое количество оперативной памяти.

Например, файл **пример.json** содержит следующий текст.

```
[
  {
    "Код" : "000000017",
    "Наименование" : "ОАО Топаз",
    "Телефоны" : [
      "+7(999)528-96-21",
      "+7(999)456-87-68"
    ],
    "ОбъемПродаж" : 5000000,
    "ЭтоПоставщик" : false
  },
  {
    "Код" : "000000006",
    "Наименование" : "АО Пантера",
    "Телефоны" : [
      "+7(999)890-987",
      "+7(999)261-79-79",
      "+7(999)789-67-85"
    ],
    "ОбъемПродаж" : 2500000,
    "ЭтоПоставщик" : true
  }
]
```

С помощью метода `ЧтениеJson.Следующий()`, используя последовательное чтение, вы позиционируетесь на начале первого объекта, описывающего партнёра.

```
Чтение.Следующий()    // начало массива
Чтение.Следующий()    // начало объекта
```

Затем, используя объектную технику, вы десериализуете всего партнёра целиком и сразу же обрабатываете его данные в методе **ОбработатьПартнера()**.

```
ОбработатьПартнера(Чтение.ПрочитатьСодержимое())
```

После этого позиция экземпляра `ЧтениеJson` (`ЧтениеJson.ТекущаяСтрока`) переместится на начало следующего объекта, описывающего партнёра. Поэтому партнёров вы десериализуете в бесконечном цикле.

```
пока Истина
    ...
;
```

Условие выхода из цикла — другой вид узла. Когда вы прочитаете последнего партнёра, `ЧтениеJson` позиционируется на последний узел, а это узел `ВидУзлаJson.КонецМассива`. В этом случае вы прерываете бесконечный цикл.

```
если Чтение.ВидУзла != ВидУзлаJson.НачалоОбъекта
    прервать
;
```

Весь пример будет выглядеть следующим образом:

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\пример.json").ОткрытьПотокЧтения()
    знч Чтение = новый ЧтениеJson(ПотокЧтения)

    Чтение.Следующий()    // начало массива
    Чтение.Следующий()    // начало объекта
    пока Истина
        ОбработатьПартнера(Чтение.ПрочитатьСодержимое())
        если Чтение.ВидУзла != ВидУзлаJson.НачалоОбъекта
            прервать
        ;
    ;
;

метод ОбработатьПартнера(Партнер: Соответствие<Строка, Объект?>)
    Консоль.Записать(Партнер.Представление())
;
```

В результате в терминал будут выведены следующие сообщения.

```
{Код: 000000017, Наименование: ОАО Топаз, Телефоны: [+7(999)528-96-21, +7(999)456-87-68], ОбъемПродаж: 5000
{Код: 000000006, Наименование: АО Пантера, Телефоны: [+7(999)890-987, +7(999)261-79-79, +7(999)789-67-85],
```

> **Примечание:**
>
> Подробнее о работе с JSON читайте в руководстве разработчика и в справке по стандартной библиотеке.

XML

Общие сведения

В настоящее время для обмена данными между информационными системами широко используется формат XML. XML (eXtensible Markup Language) — это текстовый формат обмена структурированными данными. По сравнению с JSON он более сложный и при этом обладает бóльшими возможностями.

В XML-документах используются форматирующие конструкции, по аналогии с HTML их можно назвать *тегами*. Это выражение, заключённое в угловые скобки «<» и «>». Например:

```
<section id="section_ad2_rnz_pfc">
```

Различают открывающий тег **<тег>** и закрывающий тег **</тег>**. В отличие от языка HTML, в языке XML имена тегов не фиксированы, можно давать им собственные имена, определять у них произвольные свойства с любыми типами значений.

Минимальной логической единицей XML-документа является *элемент*. Элемент начинается с открывающего тега и заканчивается закрывающим тегом. Например:

```
<title>Потоковое чтение</title>
```

Один элемент может быть вложен в другой элемент. Например:

```
<p>Список задач:
    <ul>
        <li>Первая задача</li>
        <li>Вторая задача</li>
        <li>Третья задача</li>
    </ul>
</p>
```

Здесь в элемент **p** вложен элемент **ul**, а в него вложены три элемента **li**.

XML-документ всегда имеет единственный корневой элемент, в который вложены все остальные. Между открывающим и закрывающим тегами элемента может находиться содержимое элемента. Это текст или другие вложенные элементы.

```
<КорневойЭлемент>
    <ПервыйЭлемент>Текст первого элемента</ПервыйЭлемент>
    <ВторойЭлемент>Текст второго элемента
        <ПервыйПодчиненныйВторогоЭлемента>Текст первого подчиненного</ПервыйПодчиненныйВторогоЭлемента>
        <ВторойПодчиненныйВторогоЭлемента>Текст второго подчиненного</ВторойПодчиненныйВторогоЭлемента>
    </ВторойЭлемент>
    <ТретийЭлемент>
        <ПодчиненныйЭлементТретьегоЭлемента>
            <ПодчиненныйЭлементПодчиненного>Текст элемента</ПодчиненныйЭлементПодчиненного>
        </ПодчиненныйЭлементТретьегоЭлемента>
    </ТретийЭлемент>
</КорневойЭлемент>
```

В открывающем теге элемента могут находиться *атрибуты* со своими *значениями*. Например:

```
<xref href="t000020.txt" id="u2s_1gz_pfc">Контактная информация</xref>
```

Здесь в элементе **xref** есть атрибут **href** со значением **t000020.txt** и атрибут **id** со значением **u2s_1gz_pfc**. Атрибуты содержат дополнительную информацию о том, как обрабатывать данный элемент. В этом случае содержимое элемента **xref** должно быть представлено как гиперссылка: **t000020.txt** — это имя файла, который должен быть открыт при нажатии на эту ссылку, а **u2s_1gz_pfc** — это уникальный идентификатор элемента, чтобы на него можно было сослаться из другого места XML-документа.

Запись данных в XML-документ

Для потоковой записи .xml-файла используйте тип `ЗаписьXml`. Он последовательно, от значения к значению, записывает XML в файл. Таким образом, запись XML происходит без формирования всего документа в памяти. Это является существенным преимуществом, когда объем .xml-документа может быть большим или когда он заранее неизвестен. То же самое касается и чтения .xml-файлов.

Вот пример записи информации об одном партнёре. Экземпляр `ЗаписьXml` конструируется из потока двоичных данных, в который будет выполняться запись.

```
знч Запись = новый ЗаписьXml(ПотокЗаписи)
```

Про партнёра записывается следующая информация: **Код**, **Наименование**, **Телефоны** (два телефона), **ОбъемПродаж** и булево свойство **ЭтоПоставщик**. Кроме этого записывается элемент `xref`, который представляет собой гиперссылку. Он имеет два атрибута: **href** — адрес ссылки, **id** — идентификатор элемента.

```
метод Скрипт()
    исп ПотокЗаписи = новый Файл("D:\\пример.xml").ОткрытьПотокЗаписи()
    знч Запись = новый ЗаписьXml(ПотокЗаписи)
    Запись.ЗаписатьНачалоДокумента()
        .ЗаписатьНачалоЭлемента("Партнер")
            .ЗаписатьНачалоЭлемента("Код").ЗаписатьТекст("000000017").ЗаписатьКонецЭлемента()
            .ЗаписатьНачалоЭлемента("Наименование").ЗаписатьТекст("ОАО Топаз").ЗаписатьКонецЭлемента()
            .ЗаписатьНачалоЭлемента("Телефоны")
                .ЗаписатьНачалоЭлемента("Телефон").ЗаписатьТекст("+7(999)528-96-21").ЗаписатьКонецЭлемента()
                .ЗаписатьНачалоЭлемента("Телефон").ЗаписатьТекст("+7(999)456-87-68").ЗаписатьКонецЭлемента()
            .ЗаписатьКонецЭлемента()
            .ЗаписатьНачалоЭлемента("ОбъемПродаж").ЗаписатьТекст("5000000").ЗаписатьКонецЭлемента()
            .ЗаписатьНачалоЭлемента("ЭтоПоставщик").ЗаписатьТекст("false").ЗаписатьКонецЭлемента()
            .ЗаписатьНачалоЭлемента("xref")
                .ЗаписатьАтрибут("href", "t000020.txt").ЗаписатьАтрибут("id", "u2s_1gz_pfc")
                .ЗаписатьТекст("Контактная информация")
            .ЗаписатьКонецЭлемента()
        .ЗаписатьКонецЭлемента()
    .ЗаписатьКонецДокумента()
;
```

В результате на диск **D:** будет записан файл **пример.xml** со следующим содержимым:

```
<?xml version='1.0' encoding='UTF-8'?>
<Партнер>
  <Код>000000017</Код>
  <Наименование>ОАО Топаз</Наименование>
  <Телефоны>
    <Телефон>+7(999)528-96-21</Телефон>
    <Телефон>+7(999)456-87-68</Телефон>
  </Телефоны>
  <ОбъемПродаж>5000000</ОбъемПродаж>
  <ЭтоПоставщик>false</ЭтоПоставщик>
  <xref href="t000020.txt" id="u2s_1gz_pfc">Контактная информация</xref>
</Партнер>
```

Чтение данных из XML-документа

Для потокового чтения .xml-документа используйте тип `ЧтениеXml`. Экземпляр `ЧтениеXml` конструируется из потока двоичных данных, из которого будет выполняться чтение.

```
знч Чтение = новый ЧтениеXml(ПотокЧтения)
```

Файл **пример.xml**, полученный в предыдущем примере, можно прочитать следующим образом.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\пример.xml").ОткрытьПотокЧтения()
    знч Чтение = новый ЧтениеXml(ПотокЧтения)
    пока Чтение.Следующий()
```

```
        выбор Чтение.ВидУзла
        когда НачалоЭлемента
            Консоль.Записать(Чтение.Имя)
            для индекс = 0 по Чтение.КоличествоАтрибутов() - 1
                пер ИмяАтрибута = Чтение.ИмяАтрибута(индекс)
                Консоль.Записать("    Атрибут: " + ИмяАтрибута + " = " + Чтение.ЗначениеАтрибута(ИмяАтрибута))
            ;
        когда Текст
            Консоль.Записать("    " + Чтение.Значение)
        когда КонецЭлемента
            Консоль.Записать("/")
        ;
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Партнер
Код
   000000017
/
Наименование
   ОАО Топаз
/
Телефоны
Телефон
   +7(999)528-96-21
/
Телефон
   +7(999)456-87-68
/
/
ОбъемПродаж
   5000000
/
ЭтоПоставщик
   false
/
xref
   Атрибут: href = t000020.txt
   Атрибут: id = u2s_1gz_pfc
   Контактная информация
/
/
```

Чтобы сделать вывод данных красивее, можно добавить синтаксический отступ ко вложенным элементам.

```
метод Скрипт()
    исп ПотокЧтения = новый Файл("D:\\пример.xml").ОткрытьПотокЧтения()

    пер Отступ = ""
    пер Первый = Истина
    знч Чтение = новый ЧтениеXml(ПотокЧтения)
    пока Чтение.Следующий()
        выбор Чтение.ВидУзла
        когда НачалоЭлемента
            если Первый
```

```
                Первый= Ложь
            иначе
                Отступ = Отступ + "    "
            ;
            Консоль.Записать(Отступ + Чтение.Имя)
            для индекс = 0 по Чтение.КоличествоАтрибутов() - 1
                пер ИмяАтрибута = Чтение.ИмяАтрибута(индекс)
                Консоль.Записать(Отступ + "    Атрибут: " + ИмяАтрибута + " = " + Чтение.ЗначениеАтрибута(ИмяАтри
            ;
        когда Текст
            Консоль.Записать(Отступ + "    " + Чтение.Значение)
        когда КонецЭлемента
            если Отступ.Длина() >= 3
                Отступ = Отступ.ПодстрокаСНачала(Отступ.Длина() - 3)
            ;
        ;
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Партнер
    Код
        000000017
    Наименование
        ОАО Топаз
    Телефоны
        Телефон
            +7(999)528-96-21
        Телефон
            +7(999)456-87-68
    ОбъемПродаж
        5000000
    ЭтоПоставщик
        false
    xref
        Атрибут: href = t000020.txt
        Атрибут: id = u2s_1gz_pfc
        Контактная информация
```

> **Примечание:**
>
> Подробнее о работе с XML читайте в руководстве разработчика и в справке по стандартной библиотеке.

HTTP-запросы

Общая информация

HTTP (HyperText Transfer Protocol) — это протокол передачи данных, который широко используется в Интернете. Многие информационные системы предоставляют HTTP-интерфейсы для того, чтобы другие программы могли получать данные, изменять их, добавлять и удалять. Для этого передаются HTTP-сообщения, имеющие определённую структуру.

HTTP-сообщения обычно не представляются в виде какого-то текста, состоящего из строк. Для работы с HTTP-сообщениями, как правило, используются программные объекты, оперирующие разными частями этих сообщений. HTTP-сообщения, являющиеся запросами, немного отличаются от HTTP-сообщений, являющихся ответами, но в основном они похожи.

Каждое HTTP-сообщение состоит из трёх частей:

- *Стартовая строка* — определяет тип сообщения (запрос или модификация или удаление данных и т. д.) и адрес в Интернете.
- *Заголовки* — пары ***ключ:значение***, характеризуют тело сообщения (например, кодировку текста), параметры передачи и прочие сведения.
- *Тело сообщения* — непосредственно данные сообщения. Информация, которую хотели получить (если это ответ) или данные, которые нужно передать (если это запрос).

С простейшими HTTP-сообщениями вы прекрасно знакомы. Когда в браузере вы вводите адрес и переходите на какую-то интернет-страницу, браузер отправляет простейшее HTTP-сообщение. Это GET-сообщение (запрос содержимого), без тела, с необходимыми заголовками, которые браузер сформировал автоматически. В ответном HTTP-сообщении, в его теле, браузер получает информацию, которую затем показывает вам на экране.

Методы HTTP-запросов

Метод — это тип HTTP-запроса, который находится в начале стартовой строки. Основные методы:

- GET — запрашивает данные;
- POST — передаёт данные, добавляет их;
- DELETE — удаляет данные.

Кроме этого есть методы CONNECT, HEAD, OPTIONS, PATCH, PUT и TRACE (подробнее).

URL

URL (Uniform Resource Locator) — это вторая часть стартовой строки. Это адрес в Интернете, по которому происходит обращение. Обычно URL состоит из нескольких сегментов, которые отделяются друг от друга косой чертой «/». Например:

```
https://1cmycloud.com/console/help/lang/docs/topics/1c-element-language-overview/
```

HTTP-сервисы, к которым вы будете обращаться с помощью HTTP-запросов, имеют *базовый URL* — это адрес этого сервиса в Интернете. Поскольку HTTP-сервис обычно предоставляет разную функциональность с разным набором возможных действий, для её обозначения используются разные дополнительные сегменты URL, которые добавляются к базовому URL сервиса.

Например, вы будете обращаться к HTTP-сервису с условным названием «школьный дневник». Его базовый URL **https://http-test.1cmycloud.com/applications/school/api/diary/**. Он имеет следующую функциональность:

- **/schedules/day** — показать сегодняшнее расписание — https://http-test.1cmycloud.com/applications/school/api/diary/schedules/day
- **/schedules/week** — показать расписание на неделю — https://http-test.1cmycloud.com/applications/school/api/diary/schedules/week

Также в URL могут содержаться *параметры* — дополнительная информация, необходимая для выполнения запроса. Параметры представляют собой пары ***ключ:значение*** и перечисляются в конце URL после служебного знака вопроса «?». Например:

```
https://http-test.1cmycloud.com/applications/school/api/diary/schedules/day?date=19-04-2025
```

Здесь передаётся параметр **date**, имеющий значение **19-04-2025**. Применительно к показу расписания это может означать просьбу показать расписание на указанный день **19-04-2025**.

Если параметров несколько, они отделяются друг от друга амперсандом «&». Например:

```
https://http-test.1cmycloud.com/applications/school/api/diary/schedules/week?view=past&date=12-05-2025
```

Применительно к показу оценок это может означать просьбу показать прошлое расписание (**past**) на неделю, включающую 12 мая (**12-05-2025**).

Заголовки

Заголовки — это пары ***ключ:значение***, которые содержат служебную информацию о самом сообщении, о его теле, параметры передачи и т.п. Обычно бóльшая часть заголовков создаётся автоматически. Например:

```
date : Sat, 28 Jun 2025 15:02:10 GMT
Transfer-Encoding : chunked
server : 1C
x-envoy-upstream-service-time : 7
Access-Control-Allow-Origin : *
X-Content-Type-Options : nosniff
Pragma : no-cache
Referrer-Policy : no-referrer
set-cookie : sticky-host="MTkyLjE2OC41NS4zMDoxMDA4OA=="; Path=/; HttpOnly
X-Forwarding-Server : 1C-ECRUN HTTP LoadBalancer (v2) [771-1] (aquilonis-proxy
Cache-Control : no-cache, no-store, max-age=0, must-revalidate
Vary : Origin,Access-Control-Request-Method,Access-Control-Request-Headers
Expires : 0
X-XSS-Protection : 1 ; mode=block
proxy-node-id : aquilonis-proxy-02-z1-msk4.e1c-ops.com
```

Есть некоторые заголовки, которые используются часто или бывают полезны. Например, если вы передаёте в теле запроса русские буквы и хотите увидеть их в браузере в читаемом виде, можно установить заголовок **Content-Type**:

```
Content-Type : text/plain; charset=utf-8
```

Также может быть полезен заголовок **Accept**. Наиболее распространённым форматом передачи структурированных данных в Интернете является JSON. Некоторые HTTP-сервисы могут передавать

данные и в других форматах. С помощью заголовка **Accept** вы можете указать, что хотите получить данные именно в формате JSON.

Точно так же, если вы сами передаёте в теле данные в этом формате, вы можете сообщить серверу об этом с помощью заголовка **Content-Type**. Например:

```
Accept : application/json
Content-Type : application/json
```

Тело

Тело HTTP-запроса — это набор двоичных данных, которые передаются. Это могут быть текстовые данные, структурированные данные в формате JSON, XML, двоичные данные: изображения, файлы.

Тело не является обязательным элементом HTTP-запроса. Например, GET-запросы, запрашивающие информацию с сервера, обычно не содержат тела. Все необходимые данные передаются в URL в виде параметров. Однако при работе с поисковыми системами, например Elasticsearch, условия поиска и фильтрации результатов могут быть довольно сложными, и тогда для их передачи серверу используется тело GET-запроса.

HTTP-сообщения, являющиеся ответами на запросы, обычно содержат тело. В теле передаётся запрашиваемая информация или пояснения к выполненным действиям. В то же время в некоторых случаях тела может и не быть, когда вся необходимая информация содержится в заголовках или в коде статуса: 201 (Created) — запрошенный ресурс был создан на сервере, или 204 (No Content) — запрос был успешно обработан, но нет содержимого для отправки в ответе.

Коды статуса

Код статуса является частью первой строки ответа сервера. Он представляет собой целое число из трёх цифр. За кодом статуса обычно следует поясняющая фраза на английском языке, которая разъясняет человеку причину такого ответа.

Получив ответ от сервера, всегда полезно проанализировать код статуса, чтобы определить, какие действия предпринимать дальше. Одними из самых часто встречающихся кодов являются:

- 200 OK — успешный запрос;
- 403 Forbidden — доступ запрещён;
- 404 Not Found — страница не найдена;
- 503 Service Unavailable — сервер временно не имеет возможности обрабатывать запросы по техническим причинам.

Все коды статусов разделены на 5 классов. Первая цифра указывает на класс статуса.

Последовательность отправки и получения HTTP-запроса

Для создания HTTP-запроса используйте тип `КлиентHttp`. Экземпляр этого типа можно создать с помощью одноимённого свойства глобального контекста `КлиентHttp`. Другими словами, просто нужно написать это слово в скрипте.

В простейшем случае нужно выбрать один из методов `КлиентHttp`, соответствующих методу HTTP-запроса, и указать URL. Например:

```
знч Запрос = КлиентHttp.ЗапросGet("https://http-test.1cmycloud.com/applications/school/api/diary/schedules/day
```

Теперь, если в этом есть необходимость, вы можете задать некоторые заголовки запроса. Например:

```
Запрос.УстановитьТипСодержимого("application/json").ДобавитьЗаголовок("Accept", "application/json")
```

Также можно задать тело. Например, в формате JSON передать условия отбора и фильтрации при работе с Elasticsearch. Это редкий случай, но он может понадобиться.

```
Запрос.УстановитьТело(
        "{
            \"query\": {
                \"match\": {
                    \"title\": \"Elasticsearch\"
                }
            },
            \"size\": 10,
            \"sort\": [
                {
                    \"date\": {
                        \"order\": \"desc\"
                    }
                }
            ]
        }")
```

После этого можно отправить запрос и получить результат:

```
исп Результат = Запрос.Выполнить()
```

Результат представляет собой экземпляр `ОтветHttp`. В первую очередь вас должно заинтересовать свойство `ОтветHttp.КодСтатуса`, чтобы понять, насколько успешным был запрос.

```
знч КодСтатуса = Результат.КодСтатуса
```

После этого вы можете получить тело запроса и обработать информацию, содержащуюся там.

```
пер ТелоСтрока = Результат.Тело.ПрочитатьКакСтроку()
```

Возможно, вам могут понадобиться заголовки ответа, все их можно получить, например, следующим образом:

```
для Заголовок из Результат.Заголовки.Имена()
    Консоль.Записать(Заголовок + " : " + Результат.Заголовки.ПолучитьВсе(Заголовок))
;
```

Примеры GET-запроса

На примере демонстрационного HTTP-сервиса «школьный дневник» попробуйте отправить три GET-запроса и обработать их результат. Базовый URL этого сервиса:

```
https://http-test.1cmycloud.com/applications/school/api/diary/
```

Он имеет следующую функциональность:

- /schedules/day — показать сегодняшнее расписание;
- /schedules/day/***номер_дня*** — показать расписание на день недели ***номер_дня***;
- /schedules/week — показать расписание на неделю.

Пример 1

В самом простом виде вы хотите получить сегодняшнее расписание. Отправка и обработка запроса могут выглядеть следующим образом:

```
метод Скрипт()
    исп Результат = КлиентHttp.ЗапросGet("https://http-test.1cmycloud.com/applications/school/api/diary/schedul
    Консоль.Записать(Результат.Тело.ПрочитатьКакСтроку())
;
```

В результате в терминал будут выведены следующие сообщения:

```
[
        {
                "НомерУрока": 1,
                "Предмет": "Введение в ИТ-специальность",
                "Кабинет": "П5"
        },
        {
                "НомерУрока": 2,
                "Предмет": "Введение в ИТ-специальность",
                "Кабинет": "П5"
        },
        {
                "НомерУрока": 3,
                "Предмет": "Подготовка к ГТО. 10-11 класс",
                "Кабинет": "2684"
        },
        {
                "НомерУрока": 4,
                "Предмет": "Подготовка к ГТО. 10-11 класс",
                "Кабинет": "2684"
        }
]
```

Здесь в формате JSON *(254)* находится информация о том, что сегодня 4 урока: первый и второй это **Введение в ИТ-специальность** в кабинете **П4**, а третий и четвёртый это **Подготовка к ГТО. 10-11 класс** в кабинете **2684**.

Пример 2

Второй пример — получение расписания на пятницу. Если вы планируете работать с одним HTTP-сервисом, используя разные его ресурсы, то можно создать экземпляр `КлиентHttp` с базовым адресом, а при формировании запроса просто добавлять к нему относительный путь к ресурсу.

Тело ответа можно преобразовать из формата JSON в удобное для вас представление. Как работать с JSON смотрите здесь (254).

```
метод Скрипт()
    знч DiaryКлиент = КлиентHttp.СБазовымUrl("https://http-test.1cmycloud.com/applications/school/api/diary/")
    исп Результат = DiaryКлиент.ЗапросGet("schedules/day/5").Выполнить()

    пер Уроки: неизвестно = СериализацияJson.ПрочитатьОбъект(Результат.Тело.ПрочитатьКакСтроку())
    для Урок из Уроки
        Консоль.Записать(Урок["НомерУрока"] + " " + Урок["Предмет"] + ", " + Урок["Кабинет"])
    ;
;
```

В результате в терминал будут выведены следующие сообщения.

```
1 Физическая культура, Мал. спортзал
2 Русский язык, 513
3 Литература, 524
4 Информатика, Р5
5 История, 325
6 Обществознание, 325
7 Индивидуальный проект, П7
```

Пример 3

Если нужно указать нестандартный HTTP-метод, используйте метод `КлиентHttp.СоздатьЗапрос()`. Первым параметром в него можно передать имя метода. Для GET-запроса это будет выглядеть следующим образом:

```
метод Скрипт()
    знч DiaryКлиент = КлиентHttp.СБазовымUrl("https://http-test.1cmycloud.com/applications/school/api/diary/")
    исп Результат = DiaryКлиент.СоздатьЗапрос("GET", "schedules/week").Выполнить()

    знч Чтение = новый ЧтениеJson(Результат.Тело.ПрочитатьКакСтроку())
    Чтение.Следующий()
    Чтение.Следующий()
    пока Истина
        Консоль.Записать(Чтение.Значение)
        Чтение.Следующий()
        пер Уроки: неизвестно = Чтение.ПрочитатьСодержимоеКакМассив()
        для Урок из Уроки
            Консоль.Записать("    " + Урок["НомерУрока"] + " " + Урок["Предмет"] + ", " + Урок["Кабинет"])
        ;
        если Чтение.ВидУзла != ВидУзлаJson.ИмяСвойства
            прервать
        ;
    ;
;
```

В результате в терминал будут выведены следующие сообщения:

```
Понедельник
    1 Физика, 318
    2 Физика, 318
    3 Информатика, Р5
```

```
    4 Биология, 319Б
    5 Алгебра и начала математического анализа, 213
    6 Основы безопасности и защиты Родины, 515
Вторник
    1 Алгебра и начала математического анализа, 213
    2 Русский язык, 513
    3 Химия, 214
    4 Информатика, 214
    5 Литература, 524
    6 Геометрия, 213
    7 Иностранный (английский) язык, 416
Среда
    1 Физика, 318
    2 Информатика, Р5
    3 История, 523
    4 Литература, 524
    5 Алгебра и начала математического анализа, 213
    6 Геометрия, 213
    7 Обществознание, 526
Четверг
    1 Иностранный (английский) язык, 415
    2 Физика, 318
    3 Алгебра и начала математического анализа, 213
    4 География, Б214
    5 Геометрия, 213
    6 Программирование, Р5
    7 Вероятность и статистика, 213
Пятница
    1 Физическая культура, Мал. спортзал
    2 Русский язык, 513
    3 Литература, 524
    4 Информатика, Р5
    5 История, 325
    6 Обществознание, 325
    7 Индивидуальный проект, П7
Суббота
    1 Введение в ИТ-специальность, П5
    2 Введение в ИТ-специальность, П5
    3 Подготовка к ГТО. 10-11 класс, 2684
    4 Подготовка к ГТО. 10-11 класс, 2684
Воскресенье
```

Пример отправки запросов

Вы можете потренироваться в отправке HTTP-запросов. Для этого фирма «1С» сделала демонстрационный HTTP-сервис, который возвращает в теле запроса всю информацию о полученном от вас HTTP-запросе. Таким образом вы можете сформировать запрос и увидеть, что получил сервер.

HTTP-сервис находится по адресу:

```
https://http-test.1cmycloud.com/applications/school/api/query
```

Например, вы можете сформировать запрос с параметрами, своими заголовками и телом.

```
метод Скрипт()
    знч Запрос = КлиентHttp.ЗапросGet("https://http-test.1cmycloud.com/applications/school/api/query/schedules
```

```
    Запрос.УстановитьТипСодержимого("application/json").ДобавитьЗаголовок("Accept", "application/json")
    Запрос.УстановитьТело(
        "{
            \"query\": {
                \"match\": {
                    \"title\": \"Elasticsearch\"
                }
            },
            \"size\": 10,
            \"sort\": [
                {
                    \"date\": {
                        \"order\": \"desc\"
                    }
                }
            ]
        }")
    исп Результат = Запрос.Выполнить()
    Консоль.Записать(Результат.КодСтатуса.Представление() + " " + Результат.Причина)
    Консоль.Записать(Результат.Тело.ПрочитатьКакСтроку())
;
```

В результате в терминал будут выведены следующие сообщения:

```
---------- Получен http-запрос GET

Путь к ресурсу: /schedules/week

Параметры:
   date: 12-05-2025
   view: past

Заголовки:
   x-request-id: 1743bcb0-38f0-4e31-91c8-171f860cb366
   content-length: 202
   Accept: application/json
   X-Forwarded-Proto: https
   X-Forwarded-Host: http-test.1cmycloud.com
   User-Agent: 1C Enterprise.Element
   Accept-Encoding: gzip, x-gzip, deflate
   X-Forwarded-Port: 443
   X-Concat: ___
   x-envoy-original-path: /applications/school/api/query/schedules/day?view=past&date=12-05-2025
   host: app-579342.1cmycloud.com
   X-Forwarded-For: 185.12.152.254
   proxy-node-id: aquilonis-proxy-01-z1-msk4.e1c-ops.com
   Content-Type: application/json

Тело:
{
   "query": {
       "match": {
           "title": "Elasticsearch"
       }
   },
   "size": 10,
   "sort": [
```

```
        {
            "date": {
                "order": "desc"
            }
        }
    ]
}
```

Безопасное выполнение запросов

При работе с HTTP-запросами особенное значение имеет перехват возможных исключений и анализ кодов возврата. Причины, по которым запрос не выполнился успешно, могут быть самыми разными: отсутствует интернет-соединение, произошла ошибка на сервере, вы неправильно сформировали URI, вы сформировали URI правильно, но на сервере нет такого адреса, и т. д.

Поэтому запрос нужно выполнять в инструкции `попытка` и обрабатывать `ИсключениеHttp`, которое может быть выкинуто «1С:Элементом». Кроме этого, анализируйте код статуса ответного сообщения, чтобы понимать причину неуспешного выполнения запроса.

Вот, например, как можно выполнить эти действия:

```
метод Скрипт()
    попытка
        исп Результат = КлиентHttp.ЗапросGet("https://http-test.1cmycloud.com/applications/school/api/diary/sch
        знч КлассКода = (Результат.КодСтатуса / 100).ЦелаяЧасть()
        выбор КлассКода
        когда 1
            Консоль.Записать("Информация о процессе передачи.")
        когда 2
            Консоль.Записать("Успешное выполнение.")
        когда 3
            Консоль.Записать(
                "Перенаправление. Для успешного выполнения операции необходимо сделать другой запрос.
                Адрес, по которому следует произвести запрос, может быть указан в заголовке Location.")
        когда 4
            Консоль.Записать(
                "Ошибка клиента. Запрос, отправленный на сервер, содержит ошибку.
                Тело ответа может содержать более подробное объяснение.")
        когда 5
            Консоль.Записать(
                "Ошибка сервера. Во время выполнения запроса на сервере произошла ошибка.
                Тело ответа может содержать более подробное объяснение.")
        ;
        Консоль.Записать(Результат.КодСтатуса.Представление() + " " + Результат.Причина)
        Консоль.Записать(Результат.Тело.ПрочитатьКакСтроку())
    поймать Искл: ИсключениеHttp
        Консоль.Записать("Произошла ошибка. " + Искл.Описание)
    ;
;
```

Примечание:

Подробнее о работе с HTTP-запросами читайте в руководстве разработчика и в справке по стандартной библиотеке.

Глава 8. Список терминов

- Alt-ввод *(227)*
- «1С:Предприятие.Элемент» *(9)*
- «1С:Предприятие.Элемент Скрипт» *(10)*
- «1С:Элемент» *(9)*
- Аннотация *(213)*
- Аргумент метода *(84)*
- Базовый тип *(103)*
- Булево *(53)*
- Булевы операции *(55)*
- Вызов метода *(84)*
- Выражение *(35)*
- Значение *(20)*
- Именованный аргумент *(209)*
- Имя переменной *(25)*
- Имя типа *(26)*
- Индекс символа *(169)*
- Индекс элемента *(117)*
- Инициализатор *(27)*
- Инструкция *(22)*
- Интерполяция *(167)*
- Исключение *(200)*
- Коллекция *(108)*
- Комментарий *(70)*
- Конкатенация *(42)*
- Константа *(200)*
- Конструктор *(102)*
- Контекстная подсказка *(33)*
- Контракт *(103)*
- Литерал *(22)*
- Лямбда-выражение *(218)*
- Массив *(110)*
- Метод *(81)*, Метод *(101)*
- Многомерный массив *(129)*
- Многострочная строка *(171)*
- Многострочный комментарий *(72)*
- Множество *(137)*
- Модификатор *(24)*
- Необязательный аргумент *(208)*
- Неопределено *(153)*
- Область видимости *(94)*
- Обобщённый тип *(108)*
- Объявление метода *(83)*

- Обязательный аргумент *(208)*
- Однострочный комментарий *(71)*
- Операнд *(35)*
- Операция *(35)*
- Описание типа *(25)*
- Параметры метода *(83)*
- Перегрузка метода *(212)*
- Переменная *(23)*
- Перечисление *(148)*
- Позиционный аргумент *(209)*
- Представление *(21)*
- Производный тип *(103)*
- Руководство разработчика *(19)*
- Свойство *(101)*
- Синтаксический отступ *(66)*
- Скрипт *(9)*
- Соответствие *(130)*
- Составная инструкция *(62)*
- Составной тип *(152)*
- Ссылка на метод *(219)*
- Статическая типизация *(26)*
- Статический метод *(144)*
- Структура *(140)*
- Текучий интерфейс *(246)*
- Тело лямбда-выражения *(222)*
- Тело метода *(83)*
- Тернарная операция *(54)*
- Тип *(20)*, Тип *(103)*
- Тип-одиночка *(227)*
- Точка останова *(28)*
- Функциональный тип *(216)*
- Экземпляр *(101)*
- Экранирование *(170)*

Глава 9. Список инструкций, операций и символов

Блоки кода

`метод` — определение метода *(81)*.

`область` — выделение блока кода (руководство разработчика).

Инструкции

`выбор` — проверка условия `выбор` *(79)*.

`выбросить` — выбрасывание исключения *(206)*.

`для из` — обход коллекции *(122)*.

`для по` — цикл `для по` *(73)*.

`если` — проверка условия `если` *(61)*.

`знч` — объявление переменной *(24)*.

`исключение` — объявление исключения *(204)*

`исп` — объявление переменной *(24)*.

`конст` — объявление константы *(200)*.

`пер` — объявление переменной *(24)*.

`перечисление` — объявление перечисления *(148)*.

`пока` — цикл `пока` *(128)*.

`попытка` — обработка исключений *(204)*.

`структура` — объявление структуры *(140)*.

= — инструкция присваивания *(32)*.

Операции

`и` — логическое «и» *(56)*.

`или` — логическое «или» *(56)*.

`как` — операция «как» *(153)*.

`не` — логическое «не» *(60)*.

`новый` — вызов конструктора типа *(44)*.

`это` — проверка соответствия типу *(107)*.

`это не` — проверка соответствия типу с отрицанием *(107)*.

! — настойчивая операция *(155)*.

!= — сравнение на неравенство (руководство разработчика).

${ — интерполяция *(167)*.

% — остаток от деления (руководство разработчика).

%{ — интерполяция *(167)*.

* — умножение (руководство разработчика).

** — возведение в степень (руководство разработчика).

*= — умножение с присваиванием *(40)*.

+ — сложение или конкатенация *(42)* (руководство разработчика).

+= — сложение с присваиванием *(40)*.

- — вычитание или изменение знака (руководство разработчика).

-= — вычитание с присваиванием *(40)*.

-> — лямбда-операция *(218)*.

. — обращение к свойствам и методам *(102)*.

/ — деление (руководство разработчика).

/= — деление с присваиванием *(40)*.

< — сравнение на строгое меньше (руководство разработчика).

<= — сравнение на нестрогое меньше (руководство разработчика).

== — сравнение на равенство (руководство разработчика).

> — сравнение на строгое больше (руководство разработчика).

>= — сравнение на нестрогое больше (руководство разработчика).

? — тернарная операция *(54)* или операция «Безопасный доступ» *(156)*.

?? — операция «Умолчание» *(157)*.

[ ] — обращение к элементу по индексу *(121)*, обращение к элементу по ключу *(132)*.

Ключевые слова и символы

`вниз` — используется в инструкции `для по` для указания обратного направления обхода *(128)*.

`возврат` — прерывает исполнение метода и возвращает исполнение в вызвавший код (руководство разработчика).

`иначе` — часть условной инструкции `если` *(61)* или условной инструкции `выбор` *(79)*.

`иначе если` — часть условной инструкции `если` *(61)*.

`когда` — часть условной инструкции `выбор` *(79)*.

`конструктор` — часть описания структуры или исключения *(210)*.

`метод` — часть объявления структуры *(143)* или объявления перечисления *(148)*.

`неизвестно` — тип, отключающий проверку типов компилятора *(224)*

`Неопределено` — литерал типа Неопределено *(153)*.

`ничто` — обозначает тип возвращаемого значения метода *(217)*, когда метод ничего не возвращает.

`обз` — часть объявления структуры *(140)*, обязательное поле конструктора.

`по` — используется в инструкции цикла `для по` *(73)* для обозначения конечного значения счётчика цикла.

`прервать` — используется в инструкциях для по *(73)*, для из *(122)*, пока *(128)* для завершения цикла.

`продолжить` — используется в инструкциях для по *(73)*, для из *(122)*, пока *(128)* для перехода к следующей итерации цикла.

`статический метод` — часть объявления структуры *(143)* или объявления перечисления *(148)*.

`умолчание` — часть объявления перечисления *(148)*.

`этот` — используется в описании метода перечисления *(149)*, обозначает тот элемент, для которого вызывается метод.

" — двойная кавычка, обозначает начало и конец *(22)* литерала типа `Строка`.

— знак номера, обозначает начало директивы импорта *(214)*.

& — амперсанд, обозначает начало ссылки на метод *(219)*.

' и ' — апострофы, содержат регулярное выражение *(188)*.

(и) — круглые скобки:

- содержат список аргументов *(84)* в операции вызова метода;
- определяют порядок выполнения операций *(57)* в выражениях.

-> — знак лямбда-операции *(217)*.

/* и */ — косая черта и звёздочка, обозначают начало и конец многострочного комментария *(71)*.

// — две косые черты, обозначают начало однострочного комментария *(71)*.

: — двоеточие:

- обозначает начало описания типа переменной *(25)*, параметра *(83)* или возвращаемого значения;
- разделяет ключ и значение *(131)* в литерале типа `Соответствие`.

; — точка с запятой, обозначает окончание составной инструкции (если *(61)*, для по *(73)*, для из *(122)*, выбор *(79)*) или блока кода (метод *(81)*).

< и > — угловые скобки, содержат список типов-параметров обобщённого типа *(109)*.

= — знак равенства:

- обозначает значение или выражение, инициализирующее переменную *(26)*;
- используется в инструкции цикла `для по` *(73)* для обозначения начального значения счётчика цикла.

? — вопросительный знак, в описании типов, когда перечисляются типы, обозначает тип `Неопределено` *(153)*.

@ — знак "коммерческое а", обозначает начало аннотации *(213)*.

[и] — квадратные скобки, используются в литерале типа *(112)* **Массив**.

{ и } — фигурные скобки:

- используются в литералах некоторых типов: ДатаВремя *(44)*, Соответствие *(131)*, Множество *(137)*, Версия *(27)*.
- используются в выражениях интерполяции *(167)*.

| — вертикальная черта:

- разделяет имена типов в правой части операции `это` *(107)*;
- разделяет имена типов в описаниях составных типов *(152)*;
- разделяет выражение интерполяции и форматную строку *(167)*.

Сочетания клавиш

Ctrl+Пробел — открыть контекстную подсказку.

Ctrl+Shift+Пробел — открыть контекстную подсказку к аргументам метода.

Ctrl+F5 — выполнить скрипт.

F5 — отладить скрипт, продолжить отладку.

Shift+F5 — остановить отладку.

F11 — шаг с заходом.

F10 — шаг с обходом.

Shift+F11 — шаг с выходом.

Ctrl+C — копировать в буфер обмена.

Ctrl+V — вставить из буфера обмена.

Alt+Влево два раза — вернуться к той позиции, которая была до перехода к объявлению метода.

Приёмы работы

Вызвать контекстную подсказку *(7)*.

Контекстная подсказка аргументов *(48)*.

Запустить выполнение скрипта *(8)*.

Запустить отладку скрипта *(28)*.

Остановить отладку *(32)*.

Перезапустить отладку *(63)*.

Включить точку останова *(28)*.

Посмотреть значение и тип переменной *(31)*.

Изменить значение переменной в отладке *(63)*.

Посмотреть значение выражения в отладке *(36)*.

Шаг с заходом *(30)*.

Использовать справку по стандартной библиотеке *(44)*.

Выбрать имя переменной *(23)*.

Автоматически форматировать текст *(68)*.

Выделить строку текста *(68)*.

Копировать и вставить строки кода *(76)*.

Перейти к объявлению метода *(97)*.

Шаг с выходом *(98)*.

Шаг с обходом *(99)*.

Глава 10. Решения заданий

1. Задание простое

Условие смотрите здесь *(34)*.

```
метод Скрипт()
    пер МойРост = 176
    МойРост = 180
;
```

2. Задание сложное

Условие смотрите здесь *(35)*.

```
метод Скрипт()
    пер МоеИмя = "Петя"
;
```

Букву **ё** не используют в именах. Строковый литерал нужно заключать в двойные кавычки «"».

3. Задание сложное

Условие смотрите здесь *(35)*.

```
метод Скрипт()
    пер ДомашнийАдрес = "Можайское ш., д.37, кв.36, г. Одинцово, Московская обл."
;
```

4. Задание сложное

Условие смотрите здесь *(35)*.

```
метод Скрипт()
    пер Урок1 = "Музыка"
    пер Урок2 = "Математика"
;
```

Можно назвать переменные **ПервыйУрок**, **ВторойУрок**. Нельзя назвать переменные **1Урок**, **2Урок**.

5. Задание сложное

Условие смотрите здесь *(35)*.

Телефонный номер содержит только цифры. Поэтому можно записать его как число.

```
метод Скрипт()
    пер ТелефонныйНомер = 9031234567
;
```

В то же время для удобства принято выделять в нём код города (оператора) и разделять разряды. В этом случае можно записывать его как строку.

```
метод Скрипт()
    пер ТелефонныйНомер = "(903) 123-45-67"
;
```

6. Задание сложное

Условие смотрите здесь *(35)*.

```
метод Скрипт()
    пер Оценка = "5"
    Оценка     = "4+"
;
```

Переменную **Оценка** нужно инициализировать строковым значением **"5"**. Если вы проинициализируете её числовым значением **5**, она будет иметь тип `Число` и вы не сможете записать в неё значение **"4+"**.

7. Задание простое

Количество минут, которое вам понадобится, чтобы дойти до школы. Условие смотрите здесь *(38)*.

```
метод Скрипт()
    пер СредняяСкорость    = 5
    пер РасстояниеДоШколы = 6
    пер СколькоМинут = РасстояниеДоШколы / СредняяСкорость * 60
;
```

В результате в переменной **СколькоМинут** будет значение **72**.

8. Задание сложное

Сколько раз за сутки вы сможете дойти до школы и вернуться обратно. Условие смотрите здесь *(39)*.

```
метод Скрипт()
    пер СредняяСкорость    = 5
    пер РасстояниеДоШколы = 6
    пер СколькоРаз = СредняяСкорость * 24 / РасстояниеДоШколы / 2
;
```

Расстояние, которое можно пройти за сутки, равно **СредняяСкорость * 24**. В результате в переменной **СколькоРаз** будет значение **10**.

9 Задание сложное

Сколько раз в светлое время суток вы сможете доехать на велосипеде до школы и вернуться обратно. Условие смотрите здесь *(39)*.

```
метод Скрипт()
    пер СредняяСкорость = 15
    пер РасстояниеДоШколы = 6
    пер СколькоРаз = СредняяСкорость * 13 / РасстояниеДоШколы / 2
;
```

В результате в переменной **СколькоРаз** будет значение **16.25**.

10. Задание сложное

Сколько яблок будет в вашем портфеле в среду. Условие смотрите здесь *(39)*.

```
метод Скрипт()
    // Понедельник
    пер ВПортфеле = 0
    пер ВзялИзДома = 2
    ВПортфеле = ВПортфеле + ВзялИзДома

    // Вторник
    ВзялИзДома = ВзялИзДома + 2
    ВПортфеле = ВПортфеле + ВзялИзДома

    // Среда
    ВзялИзДома = ВзялИзДома + 2
    ВПортфеле = ВПортфеле + ВзялИзДома
;
```

В результате в переменной **ВПортфеле** будет значение **12**.

11. Задание простое

Сколько времени вы проводите в школе в течение дня. Условие смотрите здесь *(41)*.

```
метод Скрипт()
    пер ДлинаУрока = 45
    пер ДлинаПеремены = 15
    пер ДлинаБольшойПеремены = 25
    пер ВсегоУроков = 6
    пер ВремяВШколе = ДлинаУрока * ВсегоУроков + ДлинаПеремены * (ВсегоУроков - 2) + ДлинаБольшойПеремены
;
```

В результате в переменной **ВремяВШколе** будет значение **355**.

12. Задание сложное

Сколько ступеней нужно пройти, чтобы подняться в квартиру. Условие смотрите здесь *(41)*.

```
метод Скрипт()
    пер СтупенейВМарше = 10
    пер СтупенейВПодъезде = 5
    пер Этаж = 7
    пер СтупенейДоКвартиры = СтупенейВМарше * 2 * (Этаж - 1) + СтупенейВПодъезде
;
```

В результате в переменной **СтупенейДоКвартиры** будет значение **125**.

13. Задание простое

Запишите фразу «Мой возраст 17 лет». Условие смотрите здесь *(42)*.

```
метод Скрипт()
    пер Возраст = "17"
```

```
    пер Результат = "Мой возраст " + Возраст + " лет"
;
```

В результате в переменной **Результат** будет значение **"Мой возраст 17 лет"**.

14. Задание сложное

В переменную **Откуда** запишите адрес отправителя. Условие смотрите здесь *(43)*.

```
метод Скрипт()
    пер Улица = "Можайское ш."
    пер НомерДома = "37"
    пер НомерКвартиры = "36"
    пер Город = "Одинцово"
    пер Область = "Московская"
    пер Откуда = Улица + ", д." + НомерДома + ", кв." + НомерКвартиры + ", г. " + Город + ", " + Область + "
;
```

В результате в переменной **Откуда** будет значение **"Можайское ш., д.37, кв.36, г. Одинцово, Московская обл."**.

15. Задание простое

Запишите начало дня 2 сентября 2025 года с помощью литерала типа `ДатаВремя`. Условие смотрите здесь *(52)*.

```
метод Скрипт()
    пер Сегодня = ДатаВремя{2025-09-02 00:00:00}
;
```

В результате в переменной **Сегодня** будет значение **2025-09-02T00:00:00.000**.

16. Задание простое

Запишите начало дня 2 сентября 2025 года с помощью конструктора типа `ДатаВремя`. Условие смотрите здесь *(52)*.

```
метод Скрипт()
    пер Сегодня = новый ДатаВремя(2025, 09, 02)
;
```

В результате в переменной **Сегодня** будет значение **2025-09-02T00:00:00.000**.

17. Задание простое

Начало следующего понедельника для произвольной даты. Условие смотрите здесь *(52)*.

```
метод Скрипт()
    пер ПроизвольнаяДата = ДатаВремя.Сейчас()
    пер НачалоПонедельника = ПроизвольнаяДата.СДнемНедели(ДеньНедели.Воскресенье).КонецДня() + 1мс
;
```

В результате в переменной **НачалоПонедельника** будет, например, значение **2025-06-16T00:00:00.000**.

18. Задание сложное

Вычислите девять утра для произвольной даты. Условие смотрите здесь *(52)*.

Вариант 1, с помощью методов типа.

```
метод Скрипт()
    пер ПроизвольнаяДата = ДатаВремя.Сейчас()
    пер ДевятьУтра = ПроизвольнаяДата.НачалоДня().ДобавитьЧасы(9)
;
```

В результате в переменной **ДевятьУтра** будет, например, значение **2025-06-14T09:00:00.000**.

Вариант 2, с помощью конструктора типа.

```
метод Скрипт()
    пер ПроизвольнаяДата = ДатаВремя.Сейчас()
    пер ДевятьУтра = новый ДатаВремя(ПроизвольнаяДата.Год, ПроизвольнаяДата.Месяц, ПроизвольнаяДата.День, 09, 0, 0
;
```

В результате в переменной **ДевятьУтра** будет, например, значение **2025-06-14T09:00:00.000**.

19 Задание сложное

Узнать продолжительность пребывания в школе. Условие смотрите здесь *(54)*.

```
метод Скрипт()
    пер Сегодня = ДеньНедели.Среда
    пер ОбычноЗанятия = Время{15:25:00} - Время{08:30:00}
    пер СегодняЗанятия = Сегодня != ДеньНедели.Среда ? ОбычноЗанятия : ОбычноЗанятия.ДобавитьЧасы(2)
;
```

В результате в переменной **ОбычноЗанятия** будет значение **06:55:00.000**, в переменной **СегодняЗанятия** будет значение **08:55:00.000**.

20. Задание простое

Чтобы по пути на работу защититься от дождя, вы берёте с собой зонт. Условие смотрите здесь *(61)*.

```
метод Скрипт()
    пер ЯИдуНаРаботу = Истина
    пер ИдетДождь = Истина
    пер НадоВзятьЗонт = ЯИдуНаРаботу и ИдетДождь
;
```

21. Задание простое

Когда выходной и когда вы болеете, вы не ходите в школу. Условие смотрите здесь *(61)*.

```
метод Скрипт()
    пер СегодняВыходной= Истина
    пер ЯБолею = Истина
    пер ЯНеИдуВШколу = СегодняВыходной или ЯБолею
;
```

22. Задание сложное

В хорошую погоду, когда у вас нет занятий, вы всегда идёте гулять. Условие смотрите здесь *(61)*.

```
метод Скрипт()
    пер СегодняВыходной = Истина
    пер СегодняПраздник = Истина
    пер ХорошаяПогода = Истина
    пер ЯИдуГулять = (СегодняВыходной или СегодняПраздник) и ХорошаяПогода
;
```

23. Задание сложное

Вы работаете только по будним дням. Условие смотрите здесь *(61)*.

```
метод Скрипт()
    пер СегодняСуббота = Истина
    пер СегодняВоскресенье = Истина
    пер ЯИдуНаРаботу = не (СегодняСуббота или СегодняВоскресенье)
;
```

24. Задание простое

Сколько часов вы можете играть с другом в указанный день. Условие смотрите здесь *(65)*.

```
метод Скрипт()
    пер НомерДняНедели = 3
    пер ВремяДляИгры: Число

    если НомерДняНедели < 6
        ВремяДляИгры = 1

    иначе
        ВремяДляИгры = 4
    ;
;
```

25. Задание простое

Номер дня недели может быть задан с ошибкой. Условие смотрите здесь *(65)*.

```
метод Скрипт()
    пер НомерДняНедели = 3
    пер ВремяДляИгры: Число
    пер Сообщение: Строка

    если НомерДняНедели < 6
        ВремяДляИгры = 1

    иначе если НомерДняНедели == 6 или НомерДняНедели == 7
        ВремяДляИгры = 4

    иначе
        Сообщение = "Неправильно задан день недели"
```

```
    ;
;
```

26. Задание сложное

Название вашей покупки в виде строки поместите в переменную. Условие смотрите здесь *(66)*.

```
метод Скрипт()
    пер ЕстьКефир = Истина
    пер ЕстьРяженка = Истина
    пер ЕстьМолоко = Истина
    пер ТекущийЧас = ДатаВремя.Сейчас().Час
    пер МояПокупка = "Ничего"

    // Супермаркет.
    если ТекущийЧас >= 9 и ТекущийЧас < 20
        если ЕстьКефир
            МояПокупка = "Кефир"

        иначе если ЕстьРяженка
            МояПокупка = "Ряженка"
        ;
    ;

    // Круглосуточный магазин.
    если МояПокупка == "Ничего" и ЕстьМолоко
        МояПокупка = "Молоко"
    ;
;
```

27. Задание простое

Количество дней в каждом месяце из второй декады. Условие смотрите здесь *(79)*.

```
метод Скрипт()
    для НомерМесяца = 4 по 6
        если НомерМесяца == 5
            Консоль.Записать("5-й месяц 31 день")
        иначе
            Консоль.Записать(НомерМесяца.ВСтроку() + "-й месяц 30 дней")
        ;
    ;
;
```

В результате в терминал будут выведены следующие строки.

```
4-й месяц 30 дней
5-й месяц 31 день
6-й месяц 30 дней
```

28. Задание сложное

Школьные оценки от самой лучшей к самой худшей. Условие смотрите здесь *(79)*.

```
метод Скрипт()
    для Счетчик = 0 по 4
        Консоль.Записать((5 - Счетчик).ВСтроку())
    ;
;
```

В результате в терминал будут выведены следующие строки.

```
5
4
3
2
1
```

29. Задание простое

Склеить строку-образец указанное количество раз. Условие смотрите здесь *(91)*.

```
метод Скрипт()
    пер ДлиннаяСтрока = РазмножитьСтроку("строка для проверки", 5)
;

метод РазмножитьСтроку(СтрокаОбразец: Строка, КоличествоКопий: Число): Строка
    пер ИтоговаяСтрока: Строка
    для Счетчик = 1 по КоличествоКопий
        ИтоговаяСтрока += СтрокаОбразец
    ;
    возврат ИтоговаяСтрока
;
```

В результате в переменной **ДлиннаяСтрока** будет значение **"строка для проверкистрока для проверкистрока для проверкистрока для проверкистрока для проверки"**.

30. Задание простое

Вывести в терминал представление даты с названием дня недели. Условие смотрите здесь *(91)*.

```
метод Скрипт()
    ВывестиДатуИДень(ДатаВремя.Сейчас())
;

метод ВывестиДатуИДень(ДатаОбразец: ДатаВремя)
    Консоль.Записать("Сегодня " + ДатаОбразец.Дата.Представление() + ", " + ДатаОбразец.ДеньНедели().Представление())
;
```

В результате в терминал будет выведено, например, **Сегодня 17.06.2025, Вторник**.

31. Задание простое

Создайте массив и запишите в него названия всех месяцев по порядку. Условие смотрите здесь *(119)*.

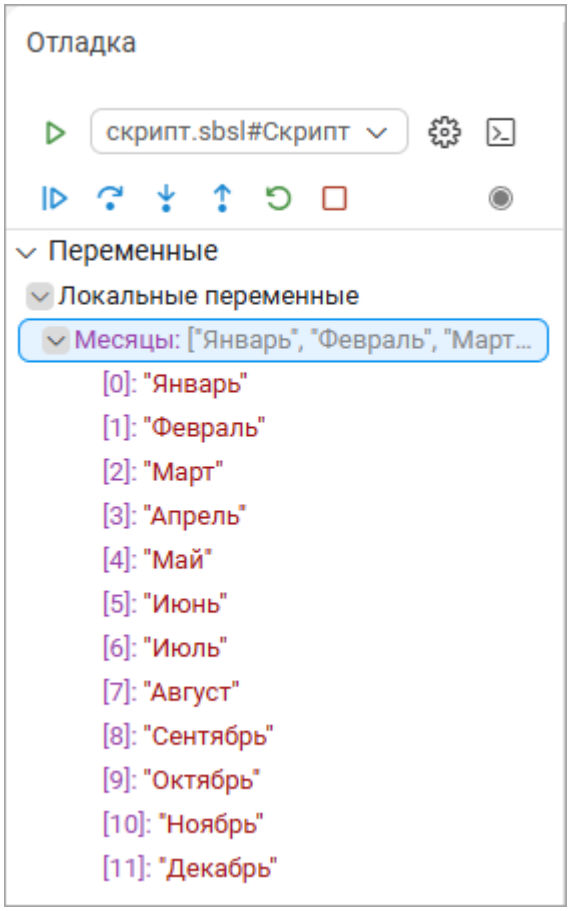
```
метод Скрипт()
    пер Месяцы = ["Январь", "Февраль", "Март",
                  "Апрель", "Май", "Июнь",
```

```
                    "Июль", "Август", "Сентябрь",
                    "Октябрь", "Ноябрь", "Декабрь"]
;
```

В результате в переменной **Месяцы** будет массив названий месяцев.

32. Задание простое

Посмотрите содержимое массива из предыдущего задания. Условие смотрите здесь *(119)*.

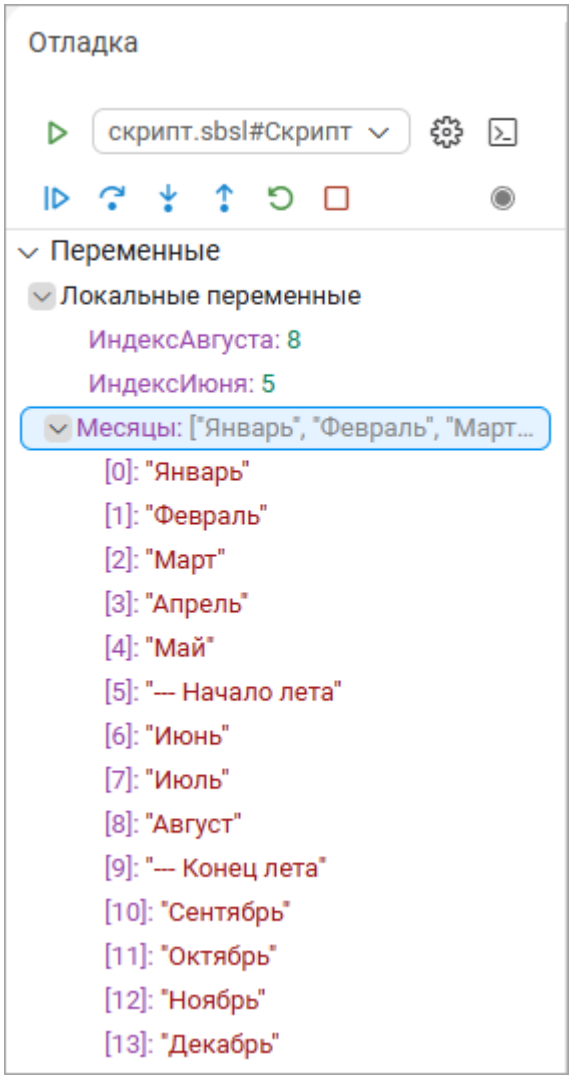


33. Задание простое

Добавьте элементы, выделяющие летние месяцы. Условие смотрите здесь *(119)*.

```
метод Скрипт()
    пер Месяцы = ["Январь", "Февраль", "Март",
                  "Апрель", "Май", "Июнь",
                  "Июль", "Август", "Сентябрь",
                  "Октябрь", "Ноябрь", "Декабрь"]
    пер ИндексИюня = Месяцы.Найти("Июнь")
    Месяцы.Вставить(ИндексИюня, "--- Начало лета")

    пер ИндексАвгуста = Месяцы.Найти("Август")
    Месяцы.Вставить(ИндексАвгуста + 1, "--- Конец лета")
;
```

В результате в переменной **Месяцы** будет следующее значение.

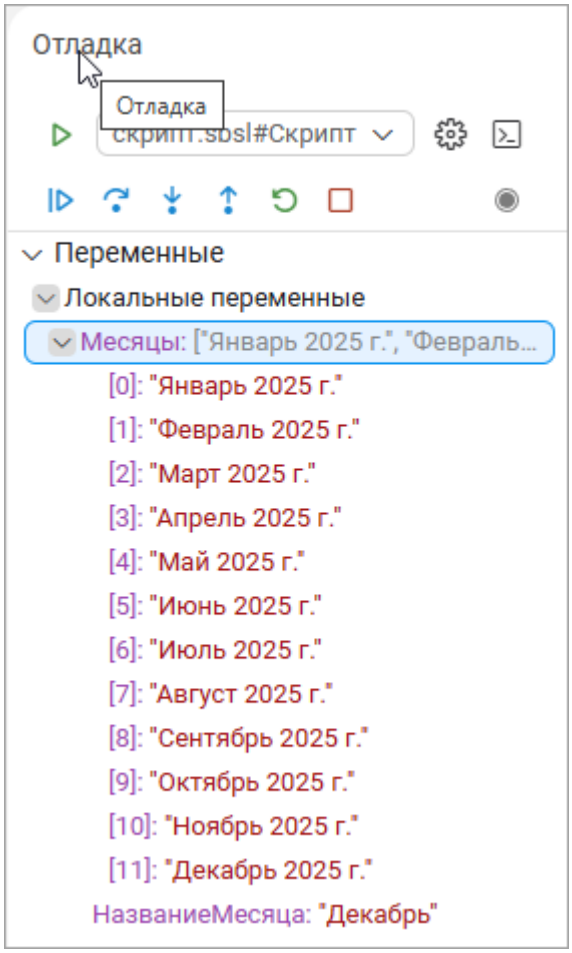


34. Задание сложное

Добавьте текущий год к каждому элементу массива. Условие смотрите здесь *(119)*.

```
метод Скрипт()
    пер Месяцы = ["Январь", "Февраль", "Март",
                  "Апрель", "Май", "Июнь",
                  "Июль", "Август", "Сентябрь",
                  "Октябрь", "Ноябрь", "Декабрь"]
    пер НазваниеМесяца: Строка
    для Индекс = 0 по Месяцы.Граница()
        НазваниеМесяца = Месяцы.Получить(Индекс)
        Месяцы.Установить(Индекс, НазваниеМесяца + " 2025 г.")
    ;
;
```

В результате в переменной **Месяцы** будет следующее значение.



35. Задание простое

Удалить ученика, который заболел. Условие смотрите здесь *(121)*.

```
метод  Скрипт()
      пер  СписокУчеников  =  ["Сергеев",  "Дмитриев",  "Захаров",  "Максимов"]
      пер  ИндексУченика  =  СпискоУчеников.Найти("Захаров")
      если  ИндексУченика  !=  Неопределено
            СпискоУчеников.УдалитьПоИндексу(ИндексУченика)
      ;
;
```

36. Задание простое

Получить названия третьего и пятого дней недели. Условие смотрите здесь *(122)*.

```
метод  Скрипт()
      пер  ДниНедели  =  ["Понедельник",  "Вторник",  "Среда",  "Четверг",
                          "Пятница",  "Суббота",  "Воскресенье"]
      пер  ТретийДеньНедели  =  ДниНедели[2]
      пер  ПятыйДеньНедели  =  ДниНедели[4]
;
```

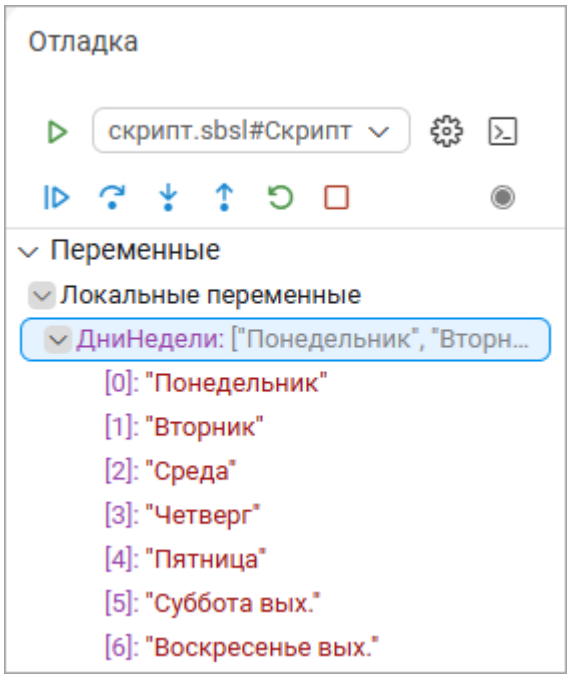
В результате в переменной **ТретийДеньНедели** будет значение **"Среда"**, в переменной **ПятыйДеньНедели** будет значение **"Пятница"**.

37. Задание простое

К выходным дням дописать « вых.». Условие смотрите здесь *(122)*.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    ДниНедели[5] = ДниНедели[5] + " вых."
    ДниНедели[6] = ДниНедели[6] + " вых."
;
```

В результате в переменной **ДниНедели** будет следующее значение.



38. Задание простое

Сложный алгоритм в теле цикла. Условие смотрите здесь *(124)*.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    для ТекущийЭлемент из ДниНедели
        выбор ТекущийЭлемент
        когда "Суббота", "Воскресенье"
            продолжить
        ;

        // Большой и сложный аргоритм
        БудниеДни.Добавить(ТекущийЭлемент)
    ;
;
```

39. Задание простое

Простой алгоритм в инструкции `если`. Условие смотрите здесь *(124)*.

```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    для ТекущийЭлемент из ДниНедели
        если не (ТекущийЭлемент == "Суббота" или ТекущийЭлемент == "Воскресенье")
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

40. Задание простое

Не обходить заведомо ненужные элементы. Условие смотрите здесь *(125)*.

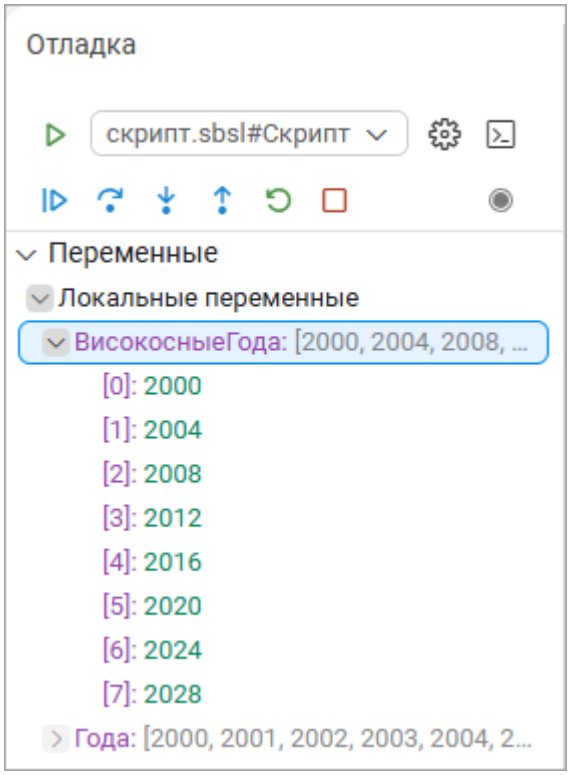
```
метод Скрипт()
    пер ДниНедели = ["Понедельник", "Вторник", "Среда", "Четверг",
                     "Пятница", "Суббота", "Воскресенье"]
    пер БудниеДни = <Строка>[]
    пер ТекущийЭлемент: Строка
    для Индекс = 0 по ДниНедели.Граница()
        ТекущийЭлемент = ДниНедели[Индекс]
        выбор ТекущийЭлемент
        когда "Суббота"
            прервать
        иначе
            БудниеДни.Добавить(ТекущийЭлемент)
        ;
    ;
;
```

41. Задание сложное

Отобрать високосные года. Условие смотрите здесь *(125)*.

```
метод Скрипт()
    пер Года = <Число>[]
    для НомерГода = 2000 по 2030
        Года.Добавить(НомерГода)
    ;

    пер ВисокосныеГода = <Число>[]
    для Год из Года
        если Год % 4 == 0
            ВисокосныеГода.Добавить(Год)
        ;
    ;
;
```

В результате в переменной **ВисокосныеГода** будет следующее значение.



В первом цикле удобно использовать `для по`, потому что в нём есть переменная, которая автоматически увеличивается на единицу. Эту переменную можно использовать в качестве номера года.

Во втором цикле удобно использовать `для из`, потому что порядок добавления високосных годов не важен, а важно только чтобы были добавлены все года, которые есть.

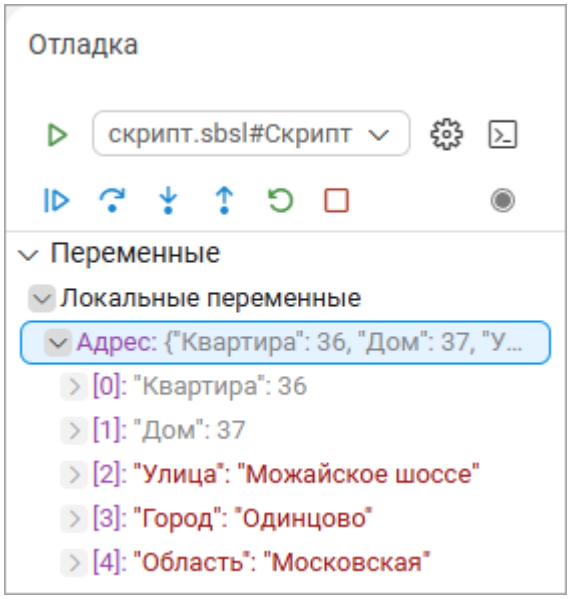
42. Задание простое

Создайте с помощью литерала структуру, в которой будет храниться ваш домашний адрес. Условие смотрите здесь *(136)*.

```
метод  Скрипт()
      пер  Адрес  =  {"Квартира":36,
                        "Дом":37,
                        "Улица":"Можайское  шоссе",
                        "Город":"Одинцово",
                        "Область":"Московская"}
;
```

43. Задание простое

Посмотрите значения отдельных элементов структуры из предыдущего примера. Условие смотрите здесь *(136)*.



44. Задание простое

Адрес вашего населённого пункта и ваш адрес в этом населённом пункте. Условие смотрите здесь *(136)*.

```
метод Скрипт()
    пер Адрес = {"Квартира":36,
                 "Дом":37,
                 "Улица":"Можайское шоссе",
                 "Город":"Одинцово",
                 "Область":"Московская"}
    пер НаселенныйПункт = Адрес["Область"] + " обл., г. " + Адрес["Город"]
    пер Квартира = Адрес["Улица"] + ", д." + Адрес["Дом"] + ", кв." + Адрес["Квартира"]
;
```

В результате в переменной **НаселенныйПункт** будет значение **Московская обл., г. Одинцово**, а в переменной **Квартира** будет значение **Можайское шоссе, д.37, кв.36**.

45. Задание простое

Создайте и заполните структуру с помощью конструктора. Условие смотрите здесь *(136)*.

```
метод Скрипт()
    пер ДниРождения = новый Соответствие<Строка, Дата>()
    ДниРождения.Вставить("Алексей", Дата{2008-10-03})
    ДниРождения.Вставить("Андрей", Дата{2008-02-29})
;
```

46. Задание сложное

Адрес может содержать корпус дома, а может и не содержать его. Условие смотрите здесь *(136)*.

```
метод Скрипт()
    пер Адрес = {"Квартира":36,
                 "Дом":37,
                 "Улица":"Можайское шоссе",
                 "Город":"Одинцово",
                 "Область":"Московская"}
```

```
    пер Корпус = Адрес.ПолучитьИлиНеопределено("Корпус")
    пер Квартира: Строка
    выбор Корпус
    когда Неопределено
        Квартира = Адрес["Улица"] + ", д." + Адрес["Дом"] + ", кв." + Адрес["Квартира"]
    иначе
        Квартира = Адрес["Улица"] + ", д." + Адрес["Дом"] + ", корп." + Корпус + ", кв." + Адрес["Квартира"]
    ;
;
```

В результате в переменной **Квартира** будет значение **Можайское шоссе, д.37, кв.36** или **Можайское шоссе, д.37, корп.3, кв.36**.

Фирма «1С»
123056, Москва, а/я 64, Селезневская ул., 21.
Тел.: (495) 737-92-57
1c@1c.ru, http://www.1c.ru/

Издательство ООО «1С-Паблишинг»
127434, Москва, Дмитровское ш., д. 9.
Тел.: (495) 681-02-21
publishing@1c.ru, http://books.1c.ru

Об опечатках просьба сообщать по адресу publishing@1c.ru.