Система 1С:Исполнитель

Содержание

Глава 1

Начало

Эта глава содержит в себе краткое описание того, что такое 1С:Исполнитель, для чего он нужен и как им пользоваться. Мы рассмотрим основные понятия этой системы, узнаем системные требования и научимся устанавливать эту систему на свой компьютер.

Что такое 1С:Исполнитель

Практически все распространенные операционные системы обладают таким понятием как "командный интерпретатор" или "интерпретатор командной строки". Это часть операционной системы, которая обеспечивает управление компьютером посредством интерактивных команд, вводимых с клавиатуры. Как правило, командный интерпретатор обладает каким-либо скриптовым языком программирования.

В Windows можно выделить интерпретатор командной строки `cmd.exe` и язык команды файлов (`.bat`-файлов). В Linux и macOS можно выделить командный интерпретатор `bash` и его язык сценариев (таких интерпретаторов много). Во всех вышеперечисленных операционных системах интерпретатор командной строки можно заменить на тот, который ближе конкретному пользователю. Например, в Windows можно установить основным командным интерпретатором `PowerShell`, а в Linux, например, установить `zsh`.

Каждый скриптовый язык обладает своими достоинствами и недостатками, но все они позволяют, как минимум, на базовом уровне автоматизировать те или иные операции, с которыми сталкиваются специалисты информационных технологий при работе с операционными системами. Эти действия можно выполнять последовательно, набирая каждую команду вручную, в интерпретаторе командной строки, а можно создать *сценарий* (или, по-другому, *скрипт*), который будет выполняться интерпретатором командной строки автоматически. Таким образом сценарий - это простой текстовый файл, содержащий программу на используемом вами языке сценариев.

Однако, если посмотреть на все вышесказанное с точки зрения специалиста, который занимается системой "1С:Предприятие", то мы сразу увидим несколько особенностей, которые надо принимать в расчет:

- Скриптовый язык должен работать во всех поддерживаемых операционных системах - он должен быть кроссплатформенным.
- Желательно, чтобы особенности каждой используемой платформы не приходилось учитывать непосредственно в самом сценарии - пусть это делает сам скриптовый язык.
- Скриптовый язык должен поддерживать работу с системой "1С:Предприятие" единым образом на всех платформах. Справедливости ради следует отметить, что это проблема не только скриптового языка.
- Крайне желательно, чтобы для использования этого средства автоматизации не приходилось учить что-то совсем новое.

Из всего вышесказанного уже становиться понятно, что такое 1С:Исполнитель. Итак, система 1С:Исполнитель это:

- Кроссплатформенный язык сценариев, который работает во всех поддерживаемых операционных системах: Linux, macOS и Windows.
- Этот язык сценариев базируется на языке системы программ "1С:Предприятие" (v8.1c.ru).
- Этот язык сценариев обладает библиотекой времени исполнения (стандартной библиотекой), очень схожей с таковой в системе программ "1С:Предприятие", поддерживая, в том числе, и очень специфические объекты. Например, доступ к кластеру серверов "1С:Предприятия".

Системные требования

Данный раздел содержит описание системных требований системы 1С:Исполнитель.

Для работы системы 1С:Исполнитель необходим компьютер, удовлетворяющий следующим требованиям:

- Операционная система:
  - Linux
  - macOS
  - Windows
- Java Runtime Envinronment (JRE) или Java Development Kit (JDK) версии 1.8 и последующие
- Процессор: Intel Core и более поздние или AMD

- Оперативная память: 4 Гбайта и более
- Дисковый накопитель: 100 Мбайт и более

Установка 1С:Исполнитель

Данный раздел содержит описание установки системы 1С:Исполнитель на компьютер пользователя.

Перед тем, как вы попытаетесь выполнить установку системы 1С:Исполнитель, убедитесь, что используется поддерживаемая операционная система, а также в том, что объем оперативной памяти компьютера и свободное пространство на дисковом накопителе соответствуют системным требованиям.

Для того, чтобы установить систему 1С:Исполнитель на ваш компьютер, выполните следующие шаги:

1. Скачайте файл, содержащий установочный комплект системы 1С:Исполнитель с официального сайта https://releases.1c.ru/project/Executor.
   На диске вашего компьютера находится файл, который содержит установочный комплект системы 1С:Исполнитель.
2. Создайте каталог, в котором будем размещаться 1С:Исполнитель.
   На диске вашего компьютера создан каталог, в котором будет размещаться файлы системы 1С:Исполнитель.
3. Скопируйте скачанный архив в каталог, где будет размещаться система 1С:Исполнитель.
4. Перейдите в каталог, где будет размещаться система 1С:Исполнитель.
5. Распакуйте архивный файл в текущий каталог.

На вашем компьютере установлена система 1С:Исполнитель и она готова к использованию.

Из чего состоит 1С:Исполнитель

Приведена структура каталога с системой 1С:Исполнитель.

Система 1С:Исполнитель представляет собой ядро для работы со скриптами на встроенном языке и консольную оболочку для запуска, передачи аргументов и вывода результатов.

В каталоге, в котором установлена система 1С:Исполнитель, находятся следующие каталоги:

- `bin` - содержит исполняемые файлы для запуска.
- `config` - конфигурационные файлы, необходимые для работы системы.
- `lib` - различные библиотеки, которые используются при работе системы 1С:Исполнитель.

Для запуск исполнения скриптов предназначены утилиты командной строки:

- Для Linux или macOS: `executor.sh`
- Для Windows: `executor.cmd`

При дальнейшем упоминании, утилита командной строки будет использоваться как executor, без указания расширения и уточнения операционной системы.

Командная строка 1С:Исполнитель

Содержит описание командной строки запуска системы 1С:Исполнитель.

`executor`
`executor` -s *имя файла* [-m *метод*] [-v] [-d *номер порта*] [--] [*параметры сценария*]

Описание параметров

-s *имя файла*

--source-path ***имя файла***

Указывает имя файла со скриптом, который будет выполняться. Скриптом считается файл, у которого расширение `.sbsl`.

Если ключ не указан, то **единственный** переданный параметр будет считаться именем файла со скриптом. Если имя запускаемого сценария не является единственным параметром при запуске `executor` (например, запускается сценарий, который получает на вход какие-то параметры), то указание параметра запуска -s является обязательным.

-m ***метод***

--method ***метод***

Указывает имя метода, с вызова которого начнется исполнение скрипта.

Если параметр не указан, то исполнение скрипта начнется с вызова метода `Скрипт` или `Script`.

-v

--V

--version

Позволяет получить версию исполнителя скриптов.

-d ***номер порта***

--debug-port ***номер порта***

Указывает номер порта, который будет использоваться сервером отладки.

Номер порта должен находиться в диапазоне от 0 до 65535 (исключая значения 0 и 65535).

--

Признак прекращения разбора аргументов командной строки. Указание этого параметра отключает разбор оставшейся части командной строки и все оставшиеся значения передаются как параметры сценария (с разделителем пробел).

Еще одним назначением параметра -- является необходимость передать в качестве значения другого параметра командной строки системы 1С:Исполнитель последовательность символов, начинающихся с `-`.

параметры сценария

Если после определения именованных параметров в командной строке запуска сценария остались какие-либо значения, то эти значения будет переданы в запускаемый сценарий в качестве параметров. Подробнее о работе параметрами запуска сценария написано здесь.

Результат работы

Во время работы исполнитель выводит информацию в стандартный поток вывода (`stdout`).

Информация об ошибках выводится в стандартный поток ошибок (`stderr`).

Кодировка выводимой информации зависит от используемой операционной системы:

- Linux: UTF-8
- macOS: UTF-8
- Windows: cp866

Результат работы скрипта сообщается в виде кода завершения исполнителя:

0

Успешное завершение работы скрипта и самого исполнителя.

255

Скрипт завершил работу с ошибкой.

Примеры запуска исполнителя

```
// 1. самый простой способ исполнить скрипт с методом "Скрипт"
executor <путь до скрипта>

// 2. аналог 1
executor -s <путь до скрипта>
executor --source-path <путь до скрипта>

// 3. исполнить скрипт с указанием метода без параметров
executor -s <путь до скрипта> -m <название метода>

// 4. исполнить скрипт с указанием метода с параметрами
executor -s <путь до скрипта> -m <название метода> <параметр метода#1>
 <параметр метода#2> ...

// 5. аналог 4 но параметры начинаются с различных символов, например "-" и
 "--"
executor -s <путь до скрипта> -m <название метода> -- -<параметр метода#1>
 --<параметр метода#2> ...

// 6. посмотреть версию исполнителя скриптов
executor -v

// 7. посмотреть версию исполнителя скриптов и исполнить скрипт
executor -v -s <путь до скрипта>

// 8. исполнить скрипт с отладкой
executor -d <порт> -s <путь до скрипта>
```

Глава 2

Описание встроенного языка

Данный раздел содержит описание языка системы 1С:Исполнитель.

Назначение и краткая характеристика языка

Приводится краткое описание языка.

Как уже было сказано выше, система 1С:Исполнитель является средой исполнения сценариев (скриптов). Целью создания системы (и языка) является максимально упростить решение задач по администрированию систем основанных на системе программ "1С:Предприятие". При этом типы данных, используемые для этого, максимально подобны аналогичным объектам, которые используются в объектной модели системы программ "1С:Предприятие". За счет этого сохраняются знания и опыт прикладных разработчиков, которые разрабатывали прикладные решения (конфигурации) в системе "1С:Предприятие". При этом сам язык, хотя и сохраняет русскоязычный синтаксис, имеет существенное количество отличий от встроенного языка системы "1С:Предприятие".

В описываемом языке, как и в любом другом языке программирования, большие программы строятся из небольшого набора базовых конструкций. В программах используются переменные, которые хранят некоторые значения. Простые выражения объединяются в более сложные с помощью операций, таких как сложение и вычитание. Значения разных типов можно помещать в коллекции, такие как массивы и множества. Выражения используются в управляющих конструкциях, таких как `если` или `для`, для управления потоком выполнения. Инструкции сгруппированы в *методы* для изоляции от прочего кода и возможности повторного использования.

Каждый объект имеет некоторое состояние, которое описывается одним или несколькими значениями. Значения, описывающие состояние, будут доступны с помощью *свойств* или *полей* объекта (термины *поле* и *свойство* могут употребляться равноправно). Для того, чтобы выполнить какое-то действие с данными объекта, предназначены методы объекта.

Блочный оператор - это такой оператор встроенного языка, который начинается с определенного ключевого слова (по которому и именуется оператор) и заканчивается символом ";"'. Блочный оператор имеет одну обязательную секцию, которая начинается сразу после имени оператора и продолжается до символа завершения (символ ";") или описания дополнительной секции. Каждая секция именуется по имени "своего" ключевого слова. Дополнительные секции могут быть не обязательными. Примеры блочных операторов и их секций:

- Оператор `попытка` может содержать дополнительные секции `поймать` и `вконце` (обработка ошибок описана здесь).
- Оператор `если` может содержать дополнительные секции `иначе если` и `иначе` (оператор `если` описан здесь).
- Оператор `выбор` может содержать дополнительные секции `когда` и `иначе` (оператор `выбор` описан здесь).
- Операторы `для`, `пока` и `область` не имеют дополнительных секций. Работа с циклами описана здесь, а про выделение области кода написано здесь.

Объявления методов, исключений, структур и перечислений также заканчиваются символом ";".

Примеры, которые будут приводиться в данной документации, будут максимально просты и не будут ставить своей задачей полноценное решение какой-то реальной проблемы. Примеры будут использоваться исключительно с целью иллюстрации работы какого-то механизма так, чтобы этот пример был максимально понятен.

Здравствуй, Мир!

```
метод Скрипт()
    Консоль.Записать("Здравствуй, Мир!")
;
```

Пространство имен

Описывается понятие "пространство имен".

Общая информация

Пространство имен - это область кода, в рамках которого система гарантирует уникальность используемых имен. Все имена принадлежат к какому-то пространству имен. Любой объект в языке имеет *квалифицированное имя*. Квалифицированное имя состоит из имени пространства имен и собственно имени сущности этого пространства имен, которые разделены символом "::" (два двоеточия). Типы данных, поставляемые вместе с языком, принадлежат стандартному пространству имен `Стд` (`Std`).

В системе 1С:Исполнитель существует пространство имен `Скрипт` (`Script`), которое включает в себя весь код, который написан в текущем файле скрипта.

Разрешение имен

Во время компиляции модуля, все неквалифицированные имена разрешаются в квалифицированные. С каждым модулем связан контекст пространств имен. Контекст пространства имен определяет, какие типы, свойства и методов доступны с указанием или без указания пространства имен.

В контексте доступна следующая информация:

- Доступные пространства имен - идентификаторы из этих пространств имен можно использовать только с указанием пространства имен.
- Импортированные пространства имен - идентификаторы из этих пространств имен можно использовать без указания пространства имен.

Если при разрешении имени (свойства, метода и типа) в контексте пространств имен какое-либо имя доступно без указания пространства имен в нескольких пространствах (в том числе в системном, собственном и импортируемом), то для разрешения имени используются приоритеты пространств имен. Пространства имен перечислены в порядке убывания приоритета:

- Пространство имен текущего сценария (`Скрипт`).
- Стандартное пространство имен (`Стд`).

Типы и их иерархия

Раздел описывает понятие тип и иерархию типов. Вводится понятие контракт.

Понятие типа и контракта

Тип - это множество допустимых значений и набор операций, которые можно выполнять над данными, принадлежащими этому типу. При описании типов будет использоваться понятие *контракт*. *Контрактом* мы будем называть набор свойств и методов, присущих тому или иному типу, а также поведение (**!**) этих свойств и методов. Другими словами контракт - это те обязательства, которые берет на себя система, когда программист оперирует значением некоторого типа. Таким образом, можно сказать, что тип - это множество допустимых значений и контракт.

Типы образуют иерархию. Это означает, что в отношении типов можно рассмотреть отношение предок-потомок. При этом потомок какого-либо типа наследует контракт своего предка, но каким-либо образом расширяет или допустимые значения или контракт или оба этих параметра. Также можно сказать, что предок какого-либо типа будет называться *базовым типом*. Соответственно потомок типа будет называться *производным типом*. У одного типа может быть несколько базовых типов. Если тип `А` выступает базовым для типа `Б`, то в переменную типа `А` можно присвоить значение типа `Б`.

В языке существует специальный тип `Тип`, который предназначен для идентификации типов значений. Этот тип необходим для определения и сравнения типов, а также для получения информации о иерархии типов.

Иерархия типов

В основании всей иерархии типов лежит тип Объект. Этот тип является базовым для всех типов, кроме типа Неопределено. У каждого из этих типов существует метод ПолучитьТип(), который возвращает значение типа Тип. Из этого следует, что у любого объекта языка можно вызвать метод ПолучитьТип(). Это хороший пример наследования контракта.

Отображение базовых типов выбранного

```
метод Скрипт()
    ПоказатьБазовыеТипы(Тип(Число))
    ПоказатьБазовыеТипы(Тип(Массив))
    ПоказатьБазовыеТипы(Тип(ПотокЗаписи))
    ПоказатьБазовыеТипы(Тип(Массив))
    ПоказатьБазовыеТипы(Тип(ИсключениеНедопустимыйАргумент))
;

метод ПоказатьБазовыеТипы(Параметр: Тип)
    Консоль.Записать("Базовые типы для типа " + Параметр)
    пер Результат: Множество = СобратьПредков(Параметр)
    для Элемент из Результат
        Консоль.ЗаписатьСтроковоеПредставление(Элемент)
    ;
;

метод СобратьПредков(Значение: Тип, Предки: Множество = {}): Множество
    для БазовыйТип из Значение.БазовыеТипы
        Предки.Добавить(БазовыйТип)
        СобратьПредков(БазовыйТип, Предки)
    ;
    возврат Предки
;
```

Переменные и работа с ними

В языке все имеет свое *имя*. Имена используются для переменных, объявляемых пользователем. С помощью имен выполняется идентификация объектов и их свойств и т.д. В данном разделе будут рассмотрены правила формирования имен, правила объявления переменных, доступ к данным, находящимися в переменных, а также особенности, связанные с видимостью имен и доступностью значений.

Имена

Имена методов, переменных, типов и прочих конструкций языка следуют следующему правилу: имя начинается с буквы (точнее со всего, что стандарт Unicode считает буквой) или с подчеркивания и может иметь сколько угодно дополнительных букв, цифр и подчеркиваний. Имена чувствительны к регистру, однако нельзя использовать имена, которые различаются только регистром. В этом смысле язык не допускает создание переменной мояПеременная и МояПеременная.

Язык имеет несколько ключевых слов, например если или возврат, которые могут использоваться только там, где разрешает синтаксис языка программирования. Кроме ключевых слов, в языке используются *литералы* - элемент программы, который непосредственно представляет значение. Самым простым примером литерала служит любое число. Другим примером литералов служит тип Булево, у которого существует всего два значения, которые представлены литералами Истина и Ложь. Ключевые слова всегда записываются в нижнем регистре. Ключевые слова и литералы не могут выступать в роли имен.

Можно использовать ключевые слова на любом из доступных языков (русский и английский). Можно произвольным образом смешивать язык написания ключевых слов.

Таблица **1.** Ключевые слова

Русский язык	Английский язык
вконце	finally
возврат	return
выбор	case
выбросить	throw
для	for
если	if
знч	val
и	and
из	in
или	or
импорт	import
иначе	else
исключение	exception
исп	use
как	as
когда	when
конст	const
конструктор	constructor
любой	any
метод	method
не	not
новый	new
область	scope
пер	var
перечисление	enumeration
по	to
поймать	catch
пока	while
попытка	try
прервать	break
продолжить	continue
структура	structure

Русский язык	Английский язык
умолчание	default
экспорт	export
это	is

Кроме ключевых слов, приведенных в таблице Таблица 1. Ключевые слова, существует некоторое количество зарезервированных слов, которые также нельзя использовать в качестве имен переменных и других создаваемых объектов.

Вернемся к литералам. В языке 1С:Исполнитель литералы могут использоваться не только для типов Число и Строка. Литералы можно использовать для создания значений типов, которые перечислены в следующей таблице.

Таблица **2.** Литералы

Литерал	Описание
"строка"	Значение типа Строка.
Байты(16-ричное-представление)	Значение типа Байты. Формат литерала приведен в описании типа.
Время(значение-времени)	Значение типа Время. Формат литерала указан в описании типа.
Дата(значение-даты)	Значение типа Дата. Формат литерала указан в описании типа.
ДатаВремя(значение-даты-времени)	Значение типа ДатаВремя. Формат литерала указан в описании типа .
значение-длительности	Значение типа Длительность. Описание формата литерала приведено в описании типа.
ИмяПеречисления.Значение	Значение перечисления. Более подробно о перечислениях написано здесь.
Истина и Ложь	Значения типа Булево.
Неопределено	Единственное значение типа Неопределено.
Тип(имя-типа)	Значение типа Тип. В качестве имя-типа необходимо написать собственно имя типа, например, для типа Строка литерал будет выглядеть следующим образом: Тип(Строка).
Уuид(значение-уuид)	Значение типа Уuид. Формат литерала приведен в описании уникального идентификатора.
ЧасоваяЗона(имя-часовой-зоны)	Значение типа ЧасоваяЗона.
Число	Значение типа Число. Формат литерала описан в описании типа.

Длина имени не ограничена. Но разработчик должен понимать, что чем выше область видимости переменной, тем более понятным должно быть это имя. Например, объявлять переменную i на уровне модуля выглядит плохой идеей, т.к. непонятно, какой смысл заложен в эту переменную. В то же время эта же переменная на уровне блока или метода выглядит вполне уместной.

Объявление переменной и типы значений

Переменные

Переменная - это ссылка на область памяти, которая хранит значение какого-либо типа. Для идентификации переменной служит *имя переменной*. На имя переменной накладываются ограничения в соответствии с правилами формирования имен языка.

Значение каждой переменной имеет какой-либо конкретный тип. Переменная может быть *простого* или *составного* типа. В первом случае в переменной может быть значение того типа, которым она объявлена (и все потомки этого типа). Если для переменной указан составной тип данных, то это означает, что переменная может принимать значение нескольких типов, которые указаны при объявлении переменной. Любая попытка присвоить переменной значение, тип которого не соответствует типу (или типам) объявления переменной будет пресекаться компилятором.

Объявление переменной может располагаться в любом месте метода. При объявлении переменной следует помнить о следующих особенностях и ограничениях:

- Не поддерживается объявление нескольких переменных в одном операторе.
- Переменная не может быть объявлена дважды в одной области видимости. Допускается объявлять переменную или описывать параметр метода так, что имя переменной или параметра будет совпадать с константой или именем свойства глобального контекста.
- Имя переменной не может совпадать с именем параметра метода.
- Переменная не может быть использована без предварительного объявления.
- Переменная не может быть использована в собственном инициализаторе.
- Не поддерживается объявление переменной без указания типа или явного значения инициализации.
- Если при объявлении переменной не указан тип, то тип будет вычислен из значения инициализации.

Объявление переменной

Для объявления переменных служит оператор объявления переменной.

```
модификатор Имя [[=Выражение] |
            [: Тип[?] [=Выражение]] |
            [: любой]]
```

модификатор

Модификатор позволяет описать некоторые характеристики объявляемой переменной:

- `пер` - создается переменная, доступна на запись и чтение.
- `знч` - создается переменная, доступная только для чтения.
- `исп` - создается переменная, тип которой должен быть потомком типа `Закрываемое`, и которая доступна только для чтения. При выходе из области видимости для такой переменной будет автоматически вызван метод `Закрыть()`.
- `конст` - позволяет определить именованное значение, которое синтаксически выглядит как переменная, но значение которого нельзя изменить после объявления. Может быть объявлена только на уровне модуля. При объявлении константы обязательно должно присутствовать значение инициализации, вычисление которого гарантировано происходит во время компиляции модуля. При инициализации литералом коллекции, значение коллекции

является неизменяемым. Правила вычислимости выражения приведены здесь.

Имя

Имя определяемой переменной. Правила формирования имени переменной см. Переменные и работа с ними.

Тип

Содержит одно или несколько имен типов, разделенных символом "|". Если определение типа переменной содержит несколько типов - происходит описание переменной составного типа. Если тип указывается как Тип?, то это равноценно объявлению составного типа Тип|Неопределено. Таким образом, если необходимо объявить переменную произвольного типа, то следует указать для нее тип Объект?.

Выражение

Выражение, которое описывает начальное значение определяемой переменной. Если выражение инициализации указывается для константы (используется модификатор конст), то это выражение должно быть вычислимо во время компиляции. Выражение является вычислимым на этапе компиляции, если оно удовлетворяет следующим критериям:

- Литералы (включая литералы коллекций) являются вычислимыми
- Обращения к константам модулей являются вычислимыми.
- Операции над вычислимыми являются вычислимыми, кроме следующих исключений:
  - Вызовы методов и обращения к свойствам.
  - Вызовы конструкторов.
  - Операции сложения значения типа Строка со значением другого типа.

Если выражение не указано, то переменная инициализируется значением по умолчанию для типа переменной. В том случае, если для выбранного типа нет значения по умолчанию или в составном типа отсутствует тип Неопределено, то во время компиляции возникает ошибка инициализации переменной.

Если в качестве типа переменной указан тип любой или тип включающий тип Неопределено, то переменная инициализируется значением Неопределено.

Если при определении переменной указано только выражение, то типом переменной будет тип результата выражения. В этом случае можно говорить о кратком объявлении переменной.

Объявляя типизированную переменную надо понимать, что во время компиляции модуля система 1С:Исполнитель будет выполнять проверку совместимости типов в выражении. Другими словами, если переменная объявлена как числовая, а ей пытаются присвоить строковое значение - это приведет к ошибке. Но

если переменная объявлена с типом любой, то во время компиляции модуля контроль совместимости типов выполняться не будет. Однако проверки времени исполнения будут выполняться в любом случае.

Объявления переменных

```
метод Скрипт()
    // Допустимые объявления переменных
    пер а: любой // Неопределено
    пер б: Число // 0
    пер в: Строка // пустая строка
    пер г: Строка? // Неопределено
    пер д: Строка|Число = 5 // 5

    // Недопустимы объявления переменных
    пер е: Строка|Число // не указано значение инициализации
    пер ж: Строка = [1,
 2] // данные инициализации не совпадают с указанным типом
;
```

Проверка соответствия типу

Оператор это проверяет, что список типов выражения является присваиваемым в список типов, перечисленных в правой части оператора.

```
ЧтоПроверяем это [не] КонтрольныйСписок
```

ЧтоПроверяем	Выражение, состав типов результата вычисления которого требуется проверить.
КонтрольныйСписок	Один или несколько типов (перечисленные через символ "|"). Оператор будет проверять присваиваемость этому списку типов. Будет выполняться проверка того, что любой из типов, входящих в состав проверяемого набора типов (ЧтоПроверяем) является потомком хотя бы одного типа из контрольного списка типов (КонтрольныйСписок) или совпадает с каким-либо типом из контрольного списка.
Результат	• Истина - тип выражения соответствует типам из СписокТипов. • Ложь - тип выражения не соответствует типам из СписокТипов.

В том случае, если необходимо проверить, что список типов выражения **не** является присваиваемым в список типов, то необходимо использовать оператор с отрицанием: *ЧтоПроверяем* это не *КонтрольныйСписок*.

Проверка на указанный тип

```
метод Скрипт()
    ЭтоКоллекция(новый Массив())
    ЭтоКоллекция(1)
    ЭтоКоллекция(новый ФиксированныйМассив([]))
    ЭтоКоллекция(новый Множество())
    ЭтоКоллекция(новый Соответствие())
;

метод ЭтоКоллекция(Параметр: Объект)
```

```
    Консоль.ЗаписатьСтроковоеПредставление("Тип параметра = "
 + Параметр.ПолучитьТип())
    если Параметр это ФиксированнаяКоллекция
        Консоль.Записать("Это коллекция!")
    иначе
        Консоль.Записать("Это не коллекция")
    ;
;
```

Приведение типов

Для того, чтобы выполнить приведение типов, предназначен оператор `как`. Данный оператор пытается привести значение выражения слева от оператора к типу справа от оператора. С помощью оператора `как` нельзя выполнить преобразование типов, другими словами, с помощью оператора `как` нельзя преобразовать значение типа `Число` к значению типа `Строка`. Приведение типов используется в тех случаях, когда значение `Выражение` может быть составного типа и в том месте, где используется приведение типов, мы хотим, чтобы выражение было одного, конкретного типа. Также приведение типов может использовать в том случае, когда типом параметра метода выступает какой-либо базовый тип (например, `Объект`). Параметр метода может выступать в качестве входного параметра другого метода, который принимает значение конкретного типа, который является потомком типа `Объект`. Тогда вначале выполняется проверка что значение нужного нам типа (например, `Строка`), а затем выполняется приведение типов при передаче в другой метод.

```
Выражение как Тип
```

Выражение	Выражение, тип результата которого необходимо привести к требуемому типу.
Тип	Тип, к которому необходимо привести тип результата вычисления *Выражение*. Этот тип должен быть одним из возможных типов для результата вычисления `Выражение`.
Результат	• Результат вычисления `Выражение`, если приведение типов возможно. • Исключение, если выполнить приведение типов невозможно.

Если возникает необходимость присвоить значение какого-то типа переменной, которая объявлена с типом `любой`, то это присваивание можно делать без использования оператора `как`.

Приведение типов

```
метод Скрипт()
    пер А: Строка|Число = ""
    пер Б: Строка|Число|Ууид = 0
    пер В: любой

    // Допустимые операции
    если А это Строка
    ;
    В = Б как Число
    А = Б как Строка|Число
    СПриведением("строка")
    СПриведением(2)

    // Недопустимые операции
    А = Б
    В = Б как ДатаВремя
;
```

```
метод СПриведением(Параметр: Объект)
    если Параметр это Строка
        МетодСтрока(Параметр как Строка)
    иначе если Параметр это Число
        МетодЧисло(Параметр как Число)
    иначе если Параметр это Булево
        МетодБулево(Параметр как Булево)
    иначе
        // действие для других типов
    ;
;
```

Присваивание значения переменной

Значение переменной задается с помощью оператора присваивания (=).

```
Назначение [АрифмОпер]= Источник
```

Назначение — В качестве назначения может выступать имя переменной или свойство объекта. Также допускается использование термина *левое значение* в качестве синонима термина *назначение*.

Источник — Выражение, значение которого будет установлено в переменную или свойство объекта. Также допускается использование термина *правое значение* в качестве синонима термина *источник*.

АрифмОпер — Язык позволяет использовать операторы присваивания, совмещенные с арифметическими операциями. Если требуется изменить Назначение на какое-то значение, то сделать это можно или с помощью выражения Назначение = Назначение + Выражение или с помощью выражения Назначение += Выражение. В качестве арифметической операции может выступать сложение (+), вычитание (-), умножение (*) и деление (/). Таким образом допустимы следующие операторы: +=, -=, *=, /=.

Область видимости имен

Область видимости это часть исходного текста программы, в пределах которой имя некоторой программной сущности (обычно — переменной, типа данных или функции), остается связанным с этой сущностью. Другими словами это та часть программы, где какое-либо имя однозначно связано с объектом системы. В рамках одной области видимости не может существовать несколько одинаковых имен. Область видимости - это свойство времени компиляции программы.

Рассмотрим пример:

```
метод Скрипт()
    пер А = Функ1()
;

метод Функ1(): Число
    пер Б: Число
    Б = 1
    возврат Б
```

```
;
```

В данном примере обратим внимание на переменную Б, объявленную в функции Функ1. *Область видимости* этой переменной - это функция Функ1 и только она. Если переменная Б будет объявлена где-то еще - это будет другая переменная, в другой области видимости.

В языке существует несколько типов областей видимости:

- Область видимости модуля. К этой области видимости относятся константы, объявленные в начале модуля. Имя переменной модуля не должно совпадать с именем ранее объявленной переменной в том же модуле.
- Область видимости блока кода. К данной области видимости относятся переменные, объявленные в теле метода в пределах блока кода. Переменная, объявленная в блоке кода имеет видимость от места объявления и до конца блока. Метод является блоком кода, параметры метода включены в его область видимости. Может быть вложенным в другую область видимости блока кода. Блоки кода образуются как различными секциями блочных операторов (если, для и т.д.), так и ключевым словом область.

Области видимости образуют иерархию. На верху иерархии находится область видимости модуля, область видимости блока, область видимости вложенного блока и т.д. В каждой области видимости доступны переменные, которые объявлены в самой области и во всех родительских областях. При этом, т.к. блоки кода могут иметь произвольную степень вложенности, то области видимости блоков кода также могут иметь произвольную степень вложенности.

Ключевое слово область предназначено для того, чтобы явным образом выделить некоторый блок кода. Области могут быть вложенными.

В качестве демонстрации различных областей видимости можно привести следующий пример:

```
метод Скрипт()
    пер Фамилия: Строка
    Фамилия = Консоль.СчитатьСтроку("Введите фамилию: ")
    // Область видимости переменной
 "Фамилия" распространяется до конца метода
    если Фамилия == "Пушкин"
        пер Книги: Массив
        // Видимые переменные: Фамилия, Книги
    иначе если Фамилия == "Бетховен"
        пер Сонаты: Массив
        // Видимые переменные: Фамилия, Сонаты
    иначе
        область
            пер СписокКомпозиторов: Массив
            // Видимые переменные: Фамилия, СписокКомпозиторов
            для Композитор из СписокКомпозиторов

 // Видимые переменные: Фамилия, СписокКомпозиторов, Композитор
            ;
            // Видимые переменные: Фамилия, СписокКомпозиторов
        ;
        область
            пер СписокКонтактов: Массив
            // Видимые переменные: Фамилия, СписокКомпозиторов
        ;
        // Видимые пременные: Фамилия
    ;
;
```

В рамках одного типа областей видимости запрещено иметь объявления, отличающиеся только регистром. В рамках разных типов областей видимости допустимы объявления с одинаковыми именами и именами, отличающимися только регистром. Приоритет имеет более вложенное (с точки зрения областей видимости)

объявление. Таким образом, становится возможным *перекрытие имен*: имя в более вложенной области может перекрыть имя, объявленное в родительской области другого типа.

```
конст СтроковаяКонстанта = "Значение инициализации"

метод Скрипт()
    пер СтроковаяКонстанта = "Перекрытие в методе возможно"
    область
        пер СтроковаяКонстанта = "Перекрытие в области уже невоможно"
    ;
;
```

Обращение к переменной

Описывает различные варианты обращения к значению переменной.

Переменную можно использовать в роли назначения в операторе присваивания. Этот момент уже рассмотрен ранее (подробнее см. Объявление переменной и типы значений).

В этом разделе мы подробнее остановимся на других способах использования переменных: доступ к свойствам и методам объектов, использовании переменной в каком-либо выражении или в роли источника в операторе присваивания.

Обращение к свойствам объекта

Если объект имеет свойства, то для доступа к свойствам используется обращение "через точку". В этом случае имя переменной и имя свойства, к которому необходимо обратиться, разделяет символ `"."`. Причем такое обращение может использоваться как для чтения значения свойства, так и для установки значения свойства (если свойство допускает изменение). Так, для обращения к свойству `Имя` объекта `ИниФайл` типа `Файл`, следует написать выражение вида `ИниФайл.Имя`.

Допускается последовательное использование операторов доступа, если получаемое значение свойства, в свою очередь, также имеет свойства, продолжая использовать объект типа `Файл`, можно написать следующий пример: `ПапкаСФайлами.Дочерние[0].Имя`. В данном примере:

- Первая точка (`ПапкаСФайлами.Дочерние`) используется для доступа к свойству `Дочерние` объекта типа `Файл`.
- Оператор `[]` используется в качестве доступа к элементу массива, который выступает в качестве одного из типов свойства `Дочерние`. В нашем примере используется доступ к самому первому элементу массива
- Последняя точка (`[0].Имя`) позволяет получить доступ к свойству `Имя` объекта типа `Файл`, который расположен в самом первом элементе массива, выступающего в качестве значения свойства `Дочерние`.

Очевидно, что такое использование чревато различными ошибками времени исполнения, поэтому такие сложные конструкции следует использовать только в том случае, когда вы полностью уверены в том, что данное выражение не приведет к ошибке.

Вызов методов объекта

Обращение к методам объектов всегда выполняется через `"."`. Вызов метода может использоваться в качестве участника выражения, если у метода есть возвращаемое значение подходящего типа. Использование вызова метода в качестве назначения оператора присваивания не поддерживается.

Система поддерживает возможность каскадного вызова методов объектов: `Наименование.Подстрока(3, 5).Длина()`. В данном примере переменная `Наименование` имеет тип `Строка`. У объекта типа `Строка` существует метод `Подстрока()`, который возвращает значение типа `Строка`. У объекта типа `Строка` существует метод `Длина()`. Таким образом, вначале из строки `Наименование` получается подстрока, а потом у получившейся подстроки получается длина. В том случае, когда вы четко не уверены, метод какого объекта будет вызван, следует использовать круглые скобки для явного определения объектов.

Выражения

Описано, что такое выражение и логическое выражение. Из чего состоят выражения и какие приоритеты у операций при вычислении выражений.

Общая информация

Выражение - конструкция, предназначенная для выполнения вычислений. Выражение является комбинацией констант, переменных и вызовов методов, связанных символами логических или арифметических операций. В качестве условия в управляющих конструкциях `если`, `пока`, `для` и операторе `?` может выступать только логическое выражение (подробнее Логические выражения).

Выражения вычисляются слева направо, с учетом приоритетов выполнения операций. Для того, чтобы избежать неоднозначности и четко понимать последовательность вычисления выражений, рекомендуется использовать круглые скобки `( )`.

Арифметические операции

В языке поддерживается стандартный набор арифметических операторов: сложение, вычитание, умножение, деление. Кроме того, поддерживаются операции возведение в степень, смена знака и получения остатка от деления. Применимость операторов выполнения арифметических действий зависит от типов операндов:

- Операция сложения: `+`. Операция применима для следующих типов:
  - `Строка + Объект? = Строка`
  - `Число + Число = Число`
  - `Длительность + Длительность = Длительность`
  - `ДатаВремя + Длительность = ДатаВремя`
  - `Дата + Длительность = Дата`
  - `Время + Длительность = Время`
  - `Момент + Длительность = Момент`
- Операция вычитания: `-`. Операция применима для следующих типов:
  - `Число - Число = Число`
  - `Длительность - Длительность = Длительность`
  - `ДатаВремя - Длительность = ДатаВремя`
  - `ДатаВремя - ДатаВремя = Длительность`
  - `Дата - Длительность = Дата`
  - `Дата - Дата = Длительность`
  - `Время - Длительность = Время`
  - `Время - Время = Длительность`
  - `Момент - Длительность = Момент`
  - `Момент - Момент = Длительность`
- Операция умножения: `*`. Операция применима для следующих типов:
  - `Число * Число = Число`
  - `Длительность * Число = Длительность`
- Операция деления: `/`. Операция применима для следующих типов:
  - `Число / Число = Число`
  - `Длительность / Число = Длительность`
- Операция получения остатка от деления: `%`. Операция применима для следующих типов:
  - `Число % Число = Число`

- Операция возведения в степень: **. Операция применима для следующих типов:
  - Число ** Число = Число
- Операция изменения знака: - (унарный минус). Операция применима для следующих типов:
  - -Число = Число
  - -Длительность = Длительность

При вычислении арифметических выражений используется следующий приоритет вычисления (в порядке уменьшения приоритета):

1. Выражения в круглых скобках.
2. Унарный минус.
3. Операции *, /, %, **.
4. Операция +, -.

Логические выражения

Выражение, результатом вычисления которого будет значение типа Булево, будет называться *логическим выражением*. В логических выражениях используются следующие операторы сравнения:

Таблица **3.** Операторы сравнения

Оператор сравнения	**Описание**
Оп1 == Оп2	Сравнение на равенство. Если значение *Оп1* равно значению *Оп2*, то результат логической операции будет значение Истина. Применимо для любых типов.
Оп1 != Оп2	Сравнение на неравенство. Если значение *Оп1* не равно значению *Оп2*, то результат логической операции будет значение Истина. Применимо для любых типов.
Оп1 > Оп2	Сравнение на строгое больше. Если значение *Оп1* строго больше *Оп2*, то результат логической операции будет значение Истина. Только для одинаковых типов, имеющих в качестве базового типа тип Сравнимое.
Оп1 >= Оп2	Сравнение на нестрогое больше. Если значение *Оп1* больше или равно *Оп2*, то результат логической операции будет значение Истина. Только для одинаковых типов, имеющих в качестве базового типа тип Сравнимое.
Оп1 < Оп2	Сравнение на строгое меньше. Если значение *Оп1* строго меньше *Оп2*, то результат логической операции будет значение Истина. Только для одинаковых типов, имеющих в качестве базового типа тип Сравнимое.
Оп1 <= Оп2	Сравнение на нестрогое больше. Если значение *Оп1* меньше или равно *Оп2*, то результат логической операции будет значение Истина. Только для одинаковых типов, имеющих в качестве базового типа тип Сравнимое.

Логические выражения могут группироваться, используя условные операторы:

Таблица **4.** Условные операторы

Условная операция	**Описание**
Выр1 и Выр2	Логическое "И". Возвращает значение Истина, когда результат вычисления каждого выражения равен Истина.
Выр1 или Выр2	Логическое "ИЛИ". Возвращает значение истина, когда результат вычисления хотя-бы одного выражения равен Истина.

Условная операция	Описание
не Выр	Логическое "НЕ". Возвращает значение Истина в том случае, если результат вычисления выражение равен Ложь.

При вычислении логических выражений используется следующий приоритета вычисления логических выражений (в порядке уменьшения приоритета):

1. Выражения в круглых скобках.
2. Операция не.
3. Операция и.
4. Операция или.

При вычислении логических выражений применяется сокращенное вычисление: вычисление логического выражения прекращается в том случае, если результат вычисления выражения полностью ясен и не может быть изменен в результате вычисления оставшейся части логического выражения.

Приведение типов в выражениях

При вычислении выражений не выполняется автоматического попытки приведения типов. Если в выражении участвуют значения несовместимых типов, то такое выражение будет или отвергнуто компилятором (если все типы, входящие в выражение известны на момент компиляции) или вычисление такого выражения завершится исключением во время исполнения (если список типов, входящих в выражение, не может быть определен во время компиляции). Если в выражении участвует значение типа любой, то и типом результата такого выражения будет любой.

Базовые типы

Данный раздел содержит описание базовых типов языка.

Тип **Число**

Тип Число представляет собой число с фиксированной точкой, произвольной точности. Точность числа ограничена 32 разрядами. Значение типа Число может быть задано литералом одного из следующих видов:

- Десятичное число: `[+|-]{0|1|2|3|4|5|6|7|8|9}[.{0|1|2|3|4|5|6|7|8|9}]`.
- Шестнадцатеричное число: `[+|-]0x{0|1|2|3|4|5|6|7|8|9|A|a|B|b|C|c|D|d|E|e|F|f}`.
- Двоичное число: `[+|-]0b{0|1}`.

Число состоит из знака числа, а также целой и дробной части, разделенных символом точка. Если задается 16-ричное или 2-ичное число, то для такого числа невозможно задание дробной части. 2- и 16-ричные числа могут быть только целыми.

Операции, определенные для значений типа Число описаны в разделе Выражения.

Работа с числами

```
метод Скрипт()
    пер А: Число
    А = 15 // целое 10-чное число
    А = 3.141592654 // дробное 10-чное число
    А = 15/7 // выражение с использованием 10-чных чисел
    А = 0xff // 16-ричное число
    А = -0xff // отрицательное 16-ричное число
    А = 0b10101 // двоичное число
;
```

Тип **Булево**

Данный раздел описывает тип, который используется для хранения логических значений.

Для хранения логических значений используется тип Булево. Данный тип имеет только два возможных значения - Истина и Ложь. Значение типа Булевоявляется типом результата вычисления логического выражения (подробнее Логические выражения). Тип Булево используется везде, где вычисляются какие-либо условия. Так, если написать выражение `а < б`, то результатом вычисления такого выражения будет значение типа Булево. Унарный оператор не представляет собой логическое отрицание, так что не Истина равно Ложь (и наоборот).

Выражение проверки на равенство (или не равенство) значению типа Булево можно записать полностью: `если А == Истина`. Однако, допустима и сокращенная запись: `если А`. Аналогичным образом можно записать проверки на равенство значению Ложь: `если не А`.

Напоминание: При использовании сокращенного написания логических выражений, всегда следует помнить о том, насколько понятно будет написанное вами выражение другому разработчику или вам же, но через какое-то время. Для гарантированного порядка вычисления частей выражения - расставляйте скобки.

Понятия, связанные с данным

Обращение к переменной
Описывает различные варианты обращения к значению переменной.

Выражения
Описано, что такое выражение и логическое выражение. Из чего состоят выражения и какие приоритеты у операций при вычислении выражений.

Тип **Строка**

Описывает понятие строки.

Значения данного типа хранят неизменную последовательность символов Unicode. При необходимости каким-то образом изменить существующую последовательность символов - создается копия оригинальной строки с выполненными изменениями.

Строку можно представить с одной стороны как последовательность символов (собственно строку) так и в виде массива. В массиве каждый символ строки выступает в виде элемента массива. Обращение к элементу массива возможно по индексу. Индекс может находиться в интервале от 0 до значения, равного длине строки, исключая это значение. Таким образом, если мы имеем строку `"пример"`, то:

- Длина строки равна 6 символам.
- Индекс первого символа равен 0 (это верно для любой не пустой строки).
- Индекс последнего символа равен 5: длина строки (в примере это 6) минус 1.

Новую строку можно создать, указав набор символов, заключенный в двойные кавычки ("): `"строка для примера"` (в виде литерала). При объявлении строки в виде литерала, нужно помнить, что:

- В строковых литералах могут использоваться управляющие последовательности, которые начинаются с символа "\" (обратная косая черта, бэкслеш). Такие управляющие последовательности предназначены для размещения внутри литерала различных специальных символов, а также произвольных символов Unicode.

Таблица **5.** Управляющие последовательности

На русском	На английском	Описание
\н	\n	**Н**овая строка
\в	\r	**В**озврат каретки
\т	\t	**Т**абуляция
\\	\\	Обратная косая черта

На русском	На английском	Описание
\"	\"	Двойные кавычки
\%	\%	Символ "%"
\юXXXXX	\uXXXXX	Произвольный символ кодировки Unicode. Код задается десятичным числом.

- Если необходимо создать литерал из многострочной строки, то это можно сделать следующими способами:

  1. Оформив один литерал на разных строках, при этом каждая подстрока (включая первую) должна начинаться с новой строки. Открывающая кавычка является первым значащим символом строки (в следующем примере цифры во второй и третьей строке приведены для удобства объяснения и не должны указываться в реальной программе):

     ```
     пер МногоСтрочноеЗначение =
             1         2
     12345678901234567890
        " первая подстрока
       вторая подстрока
           третья подстрока"
     ```

     Символ, следующий за открывающей кавычкой литерала - является первым символом получающейся строки. При этом позиция этого символа в исходном тексте будет являться левой границей всех следующих подстрок. Относительно этой границы будут формировать подстроки. В примере выше, границей строки будет символ " " в 5 позиции первой подстроки. И, следовательно, левая граница многострочной строки будет находиться в позиции 5. Во всех подстроках, кроме первой, от начала строки модуля до позиции левой границы допускаются только символы пробела.

     Каждая подстрока, кроме первой, будет начинаться с первого не пробельного символа, если он расположен левее границы или с символа, расположенного в позиции левой границы. В примере первым символом второй подстроки будет символ "в" (3 позиция), т.к. это не пробельный символ и он расположен левее левой границы. Но третья подстрока начнется с двух символом " " (позиции 5 и 6). Это связано с тем, что подстрока начинается с позиции левой границы (это позиция 5).

     Из описания литерала удаляются все пробельные символы от последнего значащего символа подстроки до конца строки модуля - во всех подстроках, кроме последней. Из последней подстроки в результирующую подстроку подпадут все символы, которые находятся между позицией левой границы и символом закрывающей кавычки (исключая сам символ кавычки).

     В качестве символа-разделителя подстрок при таком способе описания много строчной строки всегда выступает символ новой строки (`\н` или `\n`). Допустимо использовать экранируемые символы, в том числе переносы строк.

     Получившаяся в примере строка эквивалентна строке, которая будет получена в следующем примере создания многострочной строки.

  2. Указав в строке управляющую последовательность "новая строка":

     ```
     пер МногоСтрочноеЗначение = " первая подстрока\нвтораяподстрока\н
       третья подстрока"
     ```

Строковые литералы поддерживает *интерполяцию* строк. Интерполяция строк - это строка, которая может включать в себя выражения интерполяции. Выражение интерполяции - это ссылка на какую-либо переменную в текущей области видимости или выражение. Другими словами - это более понятный и удобный синтаксис для конкатенации строк, совмещенный с возможностями форматирования значений. Используя интерполяцию, нужно помнить, что формирование результирующей строки происходит в момент создания строки, которая включает выражения интерполяции.

Допускается использование двух разных синтаксиса выражения интерполяции:

- Краткий синтаксис: %Выражение. В этом случае в качестве выражения должно использоваться имя переменной, доступной в данной области видимости.
- Полный синтаксис: %{Выражение}. Полный синтаксис отличается от краткого тем, что выражение может быть сложным (не только имя переменной), и допускается использование форматной строки в том случае, если результат вычисления выражения имеет тип, одним из предков которого является тип Форматируемое. Форматная строка отделяется от выражения символом "|", вертикальная черта. Содержимое форматной строки определяется типом.

Пример использования интерполяции:

```
метод Скрипт()
    область
        // вариант 1
        пер ИмяПользователя: Строка
        пер Приветствие = "Привет, %ИмяПользователя!"
        ИмяПользователя = Консоль.СчитатьСтроку("Как тебя зовут? > ")
        Консоль.Записать(Приветствие)
    ;
    область
        // вариант 2
        пер ИмяПользователя: Строка
        ИмяПользователя = Консоль.СчитатьСтроку("Как тебя зовут? > ")
        пер Приветствие = "Привет, %ИмяПользователя!"
        Консоль.Записать(Приветствие)
    ;
;
```

В данном примере вариант 1 даст немного неожиданный результат. Вне зависимости от того, что введет пользователь представляясь системе, в консоль будет записана строка Привет, !. Это произойдет потому, что переменная Приветствие будет создана в тот момент, когда переменная ИмяПользователя заполнена значением по умолчанию (а это пустая строка). То, что будет присвоено в переменную ИмяПользователя после создания переменной Приветствие, уже не окажется влияние на значение, которое окажется в переменной Приветствие.

А вот вариант 2 приведет к желаемому результату. Если пользователь представится как Ипполит, то код из примера ответит Привет, Ипполит!. Такое поведение будет обусловлено тем, что переменная Приветствие, будет создана после того, как в переменную ИмяПользователя будет установлено нужное значение. Еще раз подчеркнем: интерполяция - это не шаблонная строка, это более удобный вариант для конкатенации строк.

Пример использования полного синтаксиса интерполяции с применением форматной строки для значения типа ДатаВремя:

```
метод Скрипт()
    пер Значение = ДатаВремя(2020-05-09)
    пер ТекущаяДата = "Сегодня %{Значение|д ММММ ггг, дддд}"
    Консоль.Записать(ТекущаяДата)
;
```

Типы для работы с датой и временем

Рассматриваются типы, которые используются для хранения значений даты и времени.

Общая информация

В языке определены несколько типов, позволяющих работать с датой и временем:

- Время - тип предназначен для работы исключительно со временем, без учета даты.
- Дата - тип предназначен для работы исключительно с датой, без учета времени.
- ДатаВремя и Момент - эти типы дают возможность использовать одновременно дату и время.

- `Длительность` - описывает временную продолжительность.

Работа с датой и временем

Любое событие, которое происходит в реальном мире, происходит в некоторый момент времени. Момент времени характеризуется датой и временем события. Когда речь идет о какой-либо встречи, момента модификации файла, вылет самолета конкретного рейса и т.д. - всегда речь идет о некоторых моментах времени. При работе с моментом времени активно используется понятие *временной зоны*. Временная зона - это географическая область, в которой установлено определенное официальное время, а также все изменения этих правил с течением времени. Разница между соседними временными зонами составляет, как правило, один час. За точку отсчета принимается нулевой (гринвичский) меридиан. Временную зону нулевого меридиана будем называть UTC, а время в остальных зонах будет отсчитываться от UTC, путем указания *смещения* (в положительных или отрицательных значениях). Положительные смещения от UTC - это временные зоны на восток от гринвичского меридиана, а отрицательные смещения - на запад. Так, для временной зоны `Europe/Moscow`, время определено как `UTC+3`. Здесь описана несколько упрощенная модель даты и времени. За более подробной информации следует обращаться к специализированным источникам.

Для работы с моментами времени предлагаются несколько типов. Первым типом является тип `Момент`. Этот тип хранит *абсолютный момент времени*. Говоря более простым языком - это дата и время, указанные для временной зоны UTC. Это позволяет располагать на одной шкале времени события, которые происходят в различных временных зонах, т.к. дата и время в конкретной временной зоне всегда имеет прямое отображение в UTC, путем корректировки на время смещения конкретной временной зоны. `Момент` удобно использовать для программных целей, но не очень удобно использовать для представления информации пользователю. Пользователь привык воспринимать дату и время в таком виде, чтобы сразу понимать, какая дата и время там, где человек сейчас находится (в своей временной зоне). Такую дату и время мы будем называть *локальной датой и временем*. Локальная дата и время - это абсолютный момент времени, приведенный к конкретной временной зоне. Для того, чтобы использовать локальную дату и время, предназначен тип `ДатаВремя`. Нужно понимать, что локальная дата и время не имеет смысла без указания конкретной временной зоны. Рассмотрим пример.

Имеется некоторое событие, которое произошло в Москве, 1 марта 2020 года в 13 часов ровно. Для этого события значение типа `Момент` будет равно `2020.03.01 10:00:00.000 Z`. Если это же локальное время представить в других временных зонах, то мы получим следующие значения:

- Временная зона UTC-3 (Дания, Бразилия): `2020.03.01 07:00:00.000`
- Временная зона UTC-1 (Азорские острова): `2020.03.01 09:00:00.000`
- Временная зона UTC+1 (Австрия, Германия): `2020.03.01 11:00:00.000`
- Временная зона UTC+3 (Россия (Москва), Белоруссия): `2020.03.01 13:00:00.000`
- Последовательность можно продолжить для других временных зон.

В этом примере видна разница между локальной датой и временем и абсолютным моментом времени. Таким образом, можно сделать следующий вывод: если нам необходимо сравнивать различные моменты времени в некоторой совпадающей системе координат - необходимо использовать значение типа `Момент` (абсолютный момент времени). Если нам необходимо отображать значение момента времени пользователю - необходимо использовать значение типа `ДатаВремя` (локальную дату и время). При этом следует помнить, что тип `ДатаВремя` не хранит временную зону, для которой сформировано это значение. Из этого следует, что сравнивать значения типа `ДатаВремя` имеет смысл только в том случае, когда оба сравниваемых значения заданы для одной временной зоне.

Значение типа `Момент` может быть создано следующим способом:

- С помощью конструктора типа, в котором составляющие даты и времени вводятся отдельными параметрами конструктора. Также в конструкторе следует указать используемую временную зону. Все параметры конструктора являются обязательными.
- С помощью литерала вида `Момент(ГГГГ.ММ.ДД ЧЧ:ИИ:СС TZ)`. Описание элементов литерала приведено далее.

- С помощью конструктора типа, в котором дата и время указывается в виде строкового литерала вида `ГГГГ.ММ.ДД ЧЧ:ИИ:СС Z`. При указании в виде литерала момент времени должен указываться приведенный к UTC. В этом литерале:
  - `ГГГГ` - значение года, 4 цифры.
  - `ММ` - значение месяца, 2 цифры. Значение может находиться в интервале от 01 до 12 (включая эти значения).
  - `ДД` - значение даты, 2 цифры. Значение может находиться в интервале от 01 до 31 (включая эти значения). Максимальное значение даты зависит от выбранного месяца и года.
  - `ЧЧ` - значение часа, 2 цифры. Значение может находиться в интервале от 0 до 23 (включая эти значения).
  - `ИИ` - значение минут, 2 цифры. Значение может находиться в интервале от 0 до 59 (включая эти значения).
  - `СС` - секунды, 2 цифры. Значение может находиться в интервале от 0 до 59 (включая эти значения).
  - `TZ` - временная зона для которой указывается момент времени. Если в качестве временной зоны указано значение `Z` - это означает, что момент времени указывается в UTC.
- С помощью конструктора типа, в который передаются значения типа `Дата`, типа `Время` и временную зону.
- Из значения типа `ДатаВремя` с помощью метода `ВМомент()`. При этом необходимо указать, для какой временной зоны было сформировано значение типа `ДатаВремя`.

Значение типа `ДатаВремя` может быть создано следующим способом:

- С помощью конструктора типа, в котором составляющие даты и времени вводятся отдельными параметрами конструктора. Все параметры конструктора (кроме миллисекунд) являются обязательными.
- С помощью литерала вида `ДатаВремя(ГГГГ.ММ.ДД ЧЧ:ИИ:СС)`. Описание элементов литерала приведено далее.
- С помощью конструктора типа, в котором дата и время указывается в виде строкового литерала вида `ГГГГ.ММ.ДД ЧЧ:ИИ:СС`. При указании в виде литерала момент времени указывается без привязки к какой-либо временной зоне. В этом литерале:
  - `ГГГГ` - значение года, 4 цифры.
  - `ММ` - значение месяца, 2 цифры. Значение может находиться в интервале от 01 до 12 (включая эти значения).
  - `ДД` - значение даты, 2 цифры. Значение может находиться в интервале от 01 до 31 (включая эти значения). Максимальное значение даты зависит от выбранного месяца и года.
  - `ЧЧ` - значение часа, 2 цифры. Значение может находиться в интервале от 0 до 23 (включая эти значения).
  - `ИИ` - значение минут, 2 цифры. Значение может находиться в интервале от 0 до 59 (включая эти значения).
  - `СС` - секунды, 2 цифры. Значение может находиться в интервале от 0 до 59 (включая эти значения).
- С помощью конструктора типа, в который передаются значения типа `Дата` и `Время`.
- Из значения типа `Момент` с помощью метода `ВДатаВремя()`. При этом необходимо указать, для какой временной зоны должно быть приведено время из значения `Момент`.

Тип `ДатаВремя` является потомком типа `Форматируемое`, а значит для значений этого типа поддерживается возможность указывать форматную строку для получения нужного представления значения данного типа. Форматная строка может состоять из следующих описателей формата:

Таблица **6.** Описание форматной строки

Описатель формата, русский	Описатель формата, английский	Описание	Пример
д	d	Номер дня месяца, без ведущего нуля	2020-03-01T07:03:30 = 1 2020-03-15T07:03:30 = 15
дд	dd	Номер дня месяца, с ведущим нулем	2020-03-01T07:03:30 = 01 2020-03-15T07:03:30 = 15
ддд	ddd	Сокращенное название дня недели	2020-03-15T07:03:30 = вс
дддд	dddd	Полное название дня недели	2020-03-15T07:03:30 = воскресенье
Д	D	Порядковый номер дня в году	2020-03-15T07:03:30 = 135
К	Q	Номер квартала	2020-03-15T07:03:30 = 1
ч	h	Номер часа в 12-часовом формате, без ведущего нуля	2020-03-15T07:03:30 = 7
чч	hh	Номер часа в 12-часовом формате, с ведущим нулем	2020-03-15T07:03:30 = 07
Ч	H	Номер часа в 24-часовом формате, без ведущего нуля	2020-03-15T07:03:30 = 7
ЧЧ	HH	Номер часа в 24-часовом формате, с ведущим нулем	2020-03-15T07:03:30 = 07
м	m	Номер минуты, без ведущего нуля	2020-03-15T07:03:30 = 3
мм	mm	Номер минуты, с ведущим нулем	2020-03-15T07:03:30 = 03
с	s	Номер секунды, без ведущего нуля	2020-03-15T07:03:05 = 5
сс	ss	Номер секунды, с ведущим нулем	2020-03-15T07:03:05 = 05
М	M	Номер месяца, без ведущего нуля	2020-03-15T07:03:30 = 3
ММ	MM	Номер месяца, без ведущего нуля	2020-03-15T07:03:30 = 03
ММММ	MMM	Сокращенное название месяца	2020-03-15T07:03:30 = мар.
ММММ	MMMM	Полное название месяца	2020-03-15T07:03:30 = март
г	y	Две последние цифры года, без ведущего нуля	2002-01-01T01:01:01 = 2
гг	yy	Две последние цифры года, с ведущим нулем	2002-01-01T01:01:01 = 02
ггг	yyy	Год в виде четырехзначного числа	2002-01-01T01:01:01 = 2002

Если необходимо, чтобы какой-либо описатель формата отображался "как есть", то можно использовать символ "'" (одинарной кавычки) для экранирования значений. Какой-либо текст считается экранированным, если этот текст расположен между двумя такими символами: `Консоль.ЗаписатьОшибку("это " + Значение.Форматировать("д' д'"))`.

Работа со временем

Данный тип (тип `Время`) позволяет работать только со временем. Например, если нам надо указать время выполнения какой-то ежедневной операции, то для указания этого значения достаточно указать только значение времени. Время учитывается с точностью до миллисекунд.

Значение типа `Время` может быть создано следующим способом:

- С помощью конструктора типа, в котором все составляющие времени вводятся отдельными параметрами конструктора. Необязательным параметров выступает только количество миллисекунд (последний параметр).
- С помощью литерала вида `Время(ЧЧ:ИИ:СС)`. Описание элементов литерала приведено далее.
- С помощью конструктора типа, в котором время указывается в виде строкового литерала вида `ЧЧ:ММ[:СС.ммм]`. В этом литерале:
  - `ЧЧ` - значение часа, 2 цифры. Может принимать значение от 0 до 23.
  - `ММ` - значение минут, 2 цифры. Может принимать значение от 0 до 59.
  - `СС.ммм` - значение секунд. Может принимать значение от 0 до 59. Если значение секунд указано в формате дробного числа, то целая часть числа становится значением секунд, а дробная часть числа означает количество миллисекунд.
- С помощью метода `Время.Сейчас()`, который возвращает текущее время на компьютере с точностью до миллисекунд.

Тип `Время` является потомком типа `Форматируемое`, а значит для значений этого типа поддерживается возможность указывать форматную строку для получения нужного представления значения данного типа. Форматная строка может состоять только из тех описателей формата, которые в таблице Таблица 6. Описание форматной строки относятся ко времени.

Работа со временем

```
метод Скрипт()
    пер НачалоРаботы = новый Время(9, 15, 0, 0)
    пер ОкончаниеРаботы = Время(18:00)
    пер ВремяОбеда = новый Время("13:00:00.500")
    пер ТекущееВремя = Время.Сейчас()
    пер ДлительностьРабочегоДня = ОкончаниеРаботы-НачалоРаботы
;
```

Работа с датой

Данный тип (тип `Дата`) позволяет работать только с датой. Так, для указания дня рождения достаточно только даты. Отпуск или командировка, как правило, тоже нужно указывать без явного указания времени наступления события.

Значение типа `Дата` может быть создано следующим способом:

- С помощью конструктора типа, в котором составляющие даты вводятся отдельными параметрами конструктора. Все параметры конструктора являются обязательными.
- С помощью литерала вида `Дата(ГГГГ.ММ.ДД)`. Описание элементов литерала приведено далее.
- С помощью конструктора типа, в котором дата указывается в виде литерала вида `ГГГГ.ММ.ДД`. В этом литерале:
  - `ГГГГ` - значение года, 4 цифры.
  - `ММ` - значение месяца, 2 цифры. Значение может находиться в интервале от 01 до 12 (включая эти значения).
  - `ДД` - значение даты, 2 цифры. Значение может находиться в интервале от 01 до 31 (включая эти значения). Максимальное значение даты зависит от выбранного месяца и года.
- С помощью метода `Дата.Сейчас()`, который возвращает текущую дату на компьютере.

Тип Дата является потомком типа Форматируемое, а значит для значений этого типа поддерживается возможность указывать форматную строку для получения нужного представления значения данного типа. Форматная строка может состоять только из тех описателей формата, которые в таблице Таблица 6. Описание форматной строки относятся к дате.

Работа с датой

```
метод Скрипт()
    пер НовыйГод = новый Дата(2020, 1, 1)
    пер Семинар = новый Дата("2020.03.01")
    пер ВосьмоеМарта = Дата(2020.03.08)
    пер ТекущаяДата = Дата.Сейчас()
;
```

Разность различных значений даты и времени

Кроме конкретных значений даты и времени, прикладное значение имеют разности таких значений. Например, интересно узнать, сколько времени прошло между созданием файла и его последней модификацией, сколько продолжалась встреча и т.д. Также возникают ситуации, когда надо к указанному моменту времени прибавить какое какой-то интервал времени. Например, когда закончится событие, которое начинается в указанный момент времени (дата и время) и длится 4 часа 15 минут. Для работы с интервалами времени предназначен тип Длительность. Этот тип хранит количество миллисекунд, описывающих некоторый интервал времени. Этот интервал может быть задан при создании значения типа Длительность или получен путем вычисления разницы между различными объектами работы с датой и временем.

Значение типа Длительность можно получить следующим способом:

- С помощью конструктора типа, в котором составляющие интервала вводятся отдельными параметрами конструктора. Обязательность указания параметров конструктора зависит от используемого конструктора.
- С помощью литерала, описывающего длительность. Литерал имеет вид [Ад][Бч][Вм][Гс][Дмс] / [Ad][Bh][Bm][Гs][Dms]. В этом литерале любой элемент может быть опущен, если соответствующее значение равно нулю. Компоненты означают следующее:
  - д / d - значение дней.
  - ч / h - значение часов.
  - м / m - значение минут.
  - с / s - значение секунд.
  - мс / ms - значение миллисекунд.
- В результате вычитания значений работы с датой и временем.

Для значений типа Длительность поддерживаются стандартные арифметические операции. Значение типа Длительность может участвовать в качестве одного из операндов в операциях сложения и вычитания для значений типа Дата, Время, Момент, ДатаВремя. Значение типа Длительность получается в результате вычитания значений типа Дата, Время, Момент, ДатаВремя.

Работа с длительностью

```
метод Скрипт()
    пер ДатаНачала = новый Дата(2020, 3, 1)
    пер ДатаОкончания = Дата(2020.03.10)
    пер ВремяНачала = новый Время(9, 15, 0)
    пер ВремяОкончания = Время(18:0:0)
    пер МоментВремени1 = новый Момент("2020/01/01 0:0:0 Z")
    пер МоментВремени2 = новый Момент("2020/12/31 12:59:59 Z")
    пер ПолтораЧаса = новый Длительность(1, 30, 0)
    пер ДваДня = 2д
    пер РезультатОперации: любой
```

```
    РезультатОперации = ДатаНачала + ДваДня
    РезультатОперации = ДатаНачала + 5д
    РезультатОперации = ДатаОкончания - ДатаНачала
    РезультатОперации = МоментВремени2 - 5д14ч30м
;
```

Тип **Неопределено**

Значение данного типа применяется, когда необходимо использовать пустое значение, не принадлежащее ни к одному другому типу. Например, такое значение изначально имеют переменные, объявленные с типом любой. Существует одно-единственное значение данного типа, задаваемое литералом Неопределено.

На этапе компиляции допускается присваивание переменной типа <Тип> значения типа <Тип>? (или, другими словами, <Тип>|Неопределено). На время компиляции также допускается вызов методов типа <Тип> от значения типа <Тип>?. Если во время исполнения в переменную типа <Тип>? попадет значение Неопределено, то выполнение программы завершится выбрасыванием исключения.

Особенность **Неопределено**

```
метод Скрипт()
    var Значение: Массив
    var Значение2: Массив|Неопределено
    var Значение3: Массив|Соответствие|Неопределено

    Значение = Значение2 // Ок, возможна ошибка времени исполнения
    Значение2.Размер() // Ок, возможно ошибка времени исполнения
    Значение2 = Значение3 // Ошибка компиляции
    Значение = Неопределено // Ошибка компиляции
;
```

Тип **Ууид**

Значения данного типа хранят значения статистически уникальных 128-битных значений. В других языках программирования для таких значений может применяться термин GUID. В качестве текстового представления уникального идентификатора используется строка, состоящая из тридцати двух 16-ричных цифр, разбитых на группы символами "дефис" ("-"). Для типа Ууид отсутствует значение по умолчанию.

Значение типа Ууид можно создать несколькими способами:

- С помощью конструктора без параметров. В этом случае будет создан новый уникальный идентификатор.
- Конструктор имеет строковой параметр, ожидающий представление уникального идентификатора.
- С помощью литерала. Если описывается константа типа Ууид, описать ее можно только таким способом.
- С помощью литерала особого вида Ууид(0). В этом случае будет создано значение по умолчанию для типа Ууид.

Примеры работы с типом **Ууид**

```
метод Скрипт()
    пер СлучайныйИдентификатор = новый Ууид()
    пер КонкретныйИдентификатор = новый Ууид("550e8400-e29b-41d4-
a716-446655440000")
    пер КонкретныйИдентификатор2 = Ууид(550e8400-e29b-41d4-
a716-446655440000)
    пер НулевойИдентификатор =  Ууид(0)
;
```

Тип **любой**

В ряде случаев необходимо описать переменную, тип которой мы не знаем в момент написания приложения. При этом нам важно, чтобы это была конкретная переменная. От этой переменной можно вызывать какие-то методы, эта переменная может принимать участие в операциях и т.д. Но в случае языка с явной типизацией переменных, мы не можем просто описать переменную, не указывая ее типа.

В данной ситуации можно попробовать объявить переменную с типом `Объект?`. Такой тип позволит присвоить нашей переменной абсолютно любое значение, т.к. тип `Объект` является базовым для любых других типов, кроме `Неопределено`. А суффикс "?" позволяет присваивать в нашу переменную и значение `Неопределено`. Но при таком описании переменной на доступен только метод `ПолучитьТип()` и все. Это произойдет потому, что компилятор "знает" только об этом методе.

Чтобы решить эту проблему, существует тип `любой`. Фактически, данный тип существует исключительно во время компиляции исходного текста программы. 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 `любой`.

Сказанное выше иллюстрирует пример.

```
метод Скрипт()
    область
        пер Значение: Объект?
        Значение = "Строка"
        Консоль.ЗаписатьСтроковоеПредставление(Значение.Длина())
    ;
    область
        пер Значение: любой
        Значение = "Строка"
        Консоль.ЗаписатьСтроковоеПредставление(Значение.Длина())
    ;
;
```

Попытка выполнить данный пример (в приведенном виде) приведет к ошибке компиляции `Вызовы неизвестного метода "Длина"` в строке `Консоль.ЗаписатьСтроковоеПредставление(Значение.Длина())` первой области. Если эту строку закомментировать, то пример выполниться успешно.

Когда можно применять данный тип?

Допустим у нас есть набор структур, который описывает файлы в файловой системе. Каждая структура содержит общий набор полей, например, имя файла и его тип. И для каждого типа файла есть специфический набор полей. При этом нам нужен метод, который будет принимать на вход любую из таких структур в качестве параметра и выполнять какие-то общие действия. В этом случае встает вопрос - каким образом типизировать параметр нашего метода? Можно рассмотреть вариант составного типа, но тогда при добавлении нового типа файла придется уточнить состав типов. Другим вариантом служит описание параметра метода с помощью типа `любой`.

Такой тип позволит внутри метода обращаться к свойствам параметра, не "обращая внимания" на фактический тип полученной структуры. При этом во время компиляции ошибок возникать не будет, но контроль наличия свойств и методов все равно будет выполняться во время выполнения модуля.

```
структура ФайлИсполнимый
    пер ИмяФайла: Строка
    пер ТипФайла: Строка
    пер Архитектура: Число
;

структура ФайлТекстовый
    пер ИмяФайла: Строка
    пер ТипФайла: Строка
    пер Кодировка: Строка
;
```

```
структура ФайлКонфигурационный
    пер ИмяФайла: Строка
    пер ТипФайла: Строка
    пер Формат: Строка
;

метод Скрипт()
    пер Файл1 = новый ФайлИсполнимый("util.exe", "exe", 32)
    пер Файл2 = новый ФайлТекстовый("untitled.txt", "txt", "utf-8")
    пер Файл3 = новый ФайлКонфигурационный("default.vrd", "cfg", "xml")
    ВыполнитьДействие(Файл1)
    ВыполнитьДействие(Файл2)
    ВыполнитьДействие(Файл3)
;

метод ВыполнитьДействие(Параметр: любой)
    пер Значение = Параметр.ТипФайла
    выбор Значение
    когда "exe"
        Консоль.Записать("Архитектура исполняемого файла =
 %{Параметр.Архитектура}")
    когда "txt"
        Консоль.Записать("Кодировка текстового файла =
 %{Параметр.Кодировка}")
    когда "cfg"
        Консоль.Записать("Формат конфигурационного файла =
 %{Параметр.Формат}")
    иначе
        Консоль.ЗаписатьОшибку("Неизвестный тип файла: "
 + Параметр.ТипФайла)
    ;
;
```

Компилятор не будет "возражать" против использования свойств от разных структур у одного параметра, т.к. параметр объявлен с типом `любой`. Но если во время исполнения в качестве параметра будет передано что-то совсем неожиданное для метода - будет исключение времени выполнения.

В заключении укажем еще два важных момента, которые характерны для типа `любой`:

- Для такого типа не существует значения типа `Тип`. Выражение `Тип(любой)` не будет пропущено компилятором.
- Для типа `любой` не существует значения.

Сравнимое

Базовый тип для объектов, которые можно сравнивать.

Тип `Сравнимое` - это базовый тип, который указывает, что потомок такого типа может участвовать в операциях сравнения: `<`, `>`, `<=`, `>=`. Если какой-либо тип имеет в числе базовых типов тип `Сравнимое`, значит значения такого типа можно сравнивать между собой.

Сравнение на равенство (`==`) и неравенство (`!=`) доступны для любых типов.

Закрываемое

Потомок данного типа удерживает системные ресурсы. Следует работать с потомками данного типа очень внимательно.

В языке существует понятие закрываемого ресурса. *Закрываемый ресурс* - это значение некоторого типа, которое получается от операционной системы. Для использования этот ресурс необходимо получить у операционной системы, а после окончания использования - вернуть. Это необходимо делать потому, что ресурсы операционной системы являются конечными. Ситуация, когда ресурс не возвращается после

использования, называется утечкой ресурсов. Для того, чтобы сделать работу с закрываемым ресурсом более простой (и чтобы минимизировать утечку ресурсов операционной системы) служит тип Закрываемое.

Закрываемое является базовым типом для типов, использующих системные ресурсы. При создании объекта, являющегося потомком типа Закрываемое, выполняется получение системного ресурса. В тот момент, когда ресурс становится ненужным, следует вызвать метод Закрыть(), который является контрактом типа Закрываемое. Во время вызова метода Закрыть() гарантируется, что ресурс будет возвращен операционной системе и будут выполнен все необходимые действия перед тем, как ресурс вернется операционной системе.

Для облегчения работы с закрываемыми объектами предназначен модификатор переменной исп. Важной особенностью такого модификатора является то, что метод Закрыть() будет вызван автоматически в тот момент, когда переменная выходит из области видимости. В этом случае очевидно, что использование ресурса закончено и его надо освободить.

Из этого утверждения есть исключение, связанное с передачей закрываемого ресурса за границу текущей области видимости. Например, какой-либо закрываемый объект создается в методе и возвращается в вызывающий метод. В этом случае вызывающий код берет на себя всю ответственность за жизненный цикл закрываемого ресурса. Такое использование усложняет контроль за ресурсами и не рекомендуется для постоянного использования.

Пример работы с закрываемым ресурсом см. Работа с закрываемыми ресурсами.

Обходимое

Базовый тип для объектов, которые можно обходить.

Тип Обходимое - это базовый тип, который указывает, что для потомка такого типа может быть применен цикл обхода коллекции для. При этом не гарантируется, что будет возможно выполнить повторный обход для значения такого типа (потомка типа Обходимое).

Форматируемое

Потомки этого типа могут использовать различные форматы для формирования строкового представления.

Тип Форматируемое - это базовый тип, который указывает, что потомок такого типа может сформировать строковое представление хранимых данных с использованием различных форматов. Для формирования такого представления служит метод Форматировать(). В качестве параметра метода выступает форматная строка. Собственно форматная строка определяется тем типом, значение которого надо представить.

Байты

Неизменяемый набор байт.

С помощью типа Байты можно хранить набор произвольных значений, заданных 16-ричными цифрами. *Байт* - это единица хранения и обработки цифровой информации. Байт состоит из 8 бит, и, следовательно, может принимать значения от 0 до 255 (включая оба значения).

Указать значение типа Байты можно литералом следующего вида: Байт(X), где X - это произвольное количество 16-ричных чисел. Литерал допускается указывать как в одну строку, так и на нескольких строках.

Работа с типом **Байты**

```
метод Скрипт()
    пер НаборБайт = Байты(4d 5a)
    пер НаборБайт2 = Байты(01
        02
        03)
    Консоль.ЗаписатьСтроковоеПредставление("Размер = " + НаборБайт.Размер()
 + ": " + НаборБайт)
    Консоль.ЗаписатьСтроковоеПредставление("Размер = " + НаборБайт2.Размер()
 + ": " + НаборБайт2)
```

```
;
```

Коллекции

Что такое коллекция? Какие бывают коллекции? Об этом написано в данном разделе.

Коллекция - это объект, который содержит, тем или иным способом, набор значений одного или различных типов, и позволяющий обращаться к этим значением. Можно выделить две группы коллекций по способу хранения данных: списки значений и словари (наборы пар "ключ-значение").

В основе всей иерархии типов, предназначенных для работы с коллекциями, лежит тип `Обходимое`. Следующими в иерархии объектов расположены типы `ФиксированнаяКоллекция` и `ФиксированноеСоответствие`. Эти типы реализуют базовый контракт, необходимый для работы со списком объектов и со словарем. Типы, которые реализуют механизмы работы с коллекциями, можно разделить на две части: те, которые не предоставляют возможность изменять свой состав (*фиксированные* коллекции) и типы, которые предоставляют возможность изменения своего состава (обычные коллекции).

В языке существуют следующие типы для работы с коллекциями:

- Массив - список элементов, без контроля уникальности. Коллекция поддерживает доступ по индексу.

  Фиксированный тип: `ФиксированнаяКоллекция` (не имеет конструктора).

  Обычный тип: `ФиксированныйМассив`, `Массив`.
- Множество - список уникальных элементов. Не поддерживается доступ по индексу.

  Базовый тип: `ФиксированнаяКоллекция` (не имеет конструктора).

  Потомки: `ФиксированноеМножество`, `Множество`.
- Соответствие - словарь, список пар "ключ-значение". В коллекции поддерживается уникальность по ключу. Поддерживается доступ по индексу.

  Базовый тип: `Обходимое` (не имеет конструктора).

  Потомки: `ФиксированноеСоотвествие`, `Соответствие`.

Дальше будут подробнее рассмотрены все и типы коллекций.

Массив

Описывает работу с массивом.

Общая информация

Массив представляет собой структуру данных, которая хранит некоторый набор значений. Элементы массива могут быть как одинакового, так и различного типа. Идентификация элементов массива выполняется по индексу. Любой массив в каждый момент времени имеет определенный размер. Размер массива можно изменять динамически, например, добавив или удалив элемент массива.

Массивы могут быть неизменяемыми и изменяемыми. Как следует из самого термина, в неизменяемом массив нельзя изменить ни значение элемента массива, ни размер самого массива. Наряду с термином "неизменяемый" может применяться термин "фиксированный". В то же время, изменяемый массив - это массив, для которого поддерживается как изменение любого элемента массива, так и изменение параметров самого массива. Изменяемый массив будет называться массивом или обычным массивом. Соответственно, тип для неизменяемого массива называется `ФиксированныйМассив`, а тип для обычного массива - `Массив`.

Создание массива

Создание массива выполняется стандартным способом:

```
метод Скрипт()
```

```
    пер ПустойМассив = новый Массив()
    пер МассивСДанными = [1, 2, 3]
    пер МассивКопия = новый Массив(МассивСДанными)
    пер ДляРаботы = новый Массив()
    ДляРаботы.Добавить(1)
    ДляРаботы.Добавить(2)
    пер Коэффициенты: Массив
    Коэффициенты = [1, 2 , 3]
;
```

Так как все, созданные в примере выше, массивы являются изменяемыми, то любой из созданных массивов можно изменить заполнить необходимыми данными с помощью встроенного языка. При создании массива копированием, в новый массив переносятся все элементы массива-источника.

Создание фиксированного массива выполняется аналогично изменяемому.

```
метод Скрипт()
    пер НеизмМассивСДанными: ФиксированныйМассив = [1, 2, 3]
    пер МатКонст = [3.141592654, 2.718281828]
    пер НеизмКонст = новый ФиксированныйМассив(МатКонст)
;
```

Элементами массива могут быть значения произвольных типов. В следующем примере приводится несколько вариантов создания массивов с различным наполнением.

```
метод Скрипт()
    пер Матрица = [[], []]
    пер СложныйМассив = [ Неопределено, 1, [], ["один", "два"]]
    пер Матрица2: Массив
    Матрица2.Добавить(новый Массив())
    Матрица2.Добавить(новый Массив())
;
```

Во всех приведенных примерах создавались переменные, которые хранили ссылки на массивы значений. Однако, язык предоставляет возможность использовать массив, не объявляя для этого отдельную переменную. Для этого можно использовать специальный синтаксис `[элемент0, ..., элементN]`, где каждый элемент - это выражение на встроенном языке.

```
метод Скрипт()
    для Расширение из ["*.epf", "*.erf"]
        Консоль.Записать("Текущее расширение - " + Расширение)
        // какое-то полезное действие
    ;
    ВывестиМассив([3.141592654, 2.718281828])
;

метод ВывестиМассив(ЧтоВыводим: ФиксированныйМассив)
    для Элемент из ЧтоВыводим
        Консоль.Записать(Элемент)
    ;
;
```

Использование массива

Для обращения к элементу массива используется *индекс*. Индекс массива является целым числом, значение которого может находиться в диапазоне от 0 до 2^{32}. Массив из N элементов индексируется от 0 до N-1, где 0 - это индекс первого элемента в массиве. Обращение по индексу, указывающем за границу массива, приведет к исключению. Если значение индекса является дробным числом (не целочисленным), то обращение по такому индексу приведет к ошибке.

Для обращения к элементам массива можно использовать один из следующих способов:

- Операторный способ. В этом случае индекс требуемого элемента массива нужно указать внутри квадратных скобок `[]`. Обращение к 5 элементу массива `Коэффициенты` будет выглядеть следующим образом: `Коэффициенты[4]`.
- Функциональный способ. В этом случае индекс нужного элемента массива нужно указать в качестве параметра метода `Получить()` соответствующего массива. Обращение к 5 элементу массива `Коэффициенты` будет выглядеть следующим образом: `Коэффициенты.Получить(4)`.

```
метод Скрипт()
    пер Данные: Массив = [1, 2, 3]
    Консоль.Записать(Данные[0])
    Консоль.Записать(Данные.Получить(1))
;
```

При необходимости перебрать все элементы массива, можно воспользоваться любым вариантом организации цикла. Такие примеры будут приведены далее.

В цикле обхода коллекции не поддерживается изменение состава коллекции. При попытке такого изменения будет выброшено исключение.

```
метод Скрипт()
    пер Данные: Массив = [1, 2, 3]
    // обход циклом "для"
    для Индекс = 0 по Данные.Граница()
        Консоль.Записать(Данные.Получить(Индекс))
    ;
    // обход с помощью итератора
    для Элемент из Данные
        Консоль.Записать(Элемент)
    ;
    // обход циклом "пока"
    пер Индекс = 0
    пока Индекс <= Данные.Граница()
        Консоль.Записать(Данные.Получить(Индекс))
        Индекс += 1
    ;
;
```

Теперь разберемся, каким образом следует пользоваться массивами в том случае, когда массив является параметром метода или возвращаемым значением. Для типа формального параметры метода следует использовать тип `ФиксированныйМассив`. В этом случае в качестве параметра можно передать как фиксированный, так и обычный массив.

```
метод Скрипт()
    пер МатКонст = [3.141592654, 2.718281828]
    МетодСМассивом(МатКонст)
    МетодСМассивом(новый ФиксированныйМассив(МатКонст))
    МетодСМассивом([7, 6, 5, 4, 3, 2])
;

метод МетодСМассивом(ПараметрМассив: ФиксированныйМассив)
    Консоль.ЗаписатьСтроковоеПредставление(ПараметрМассив.ПолучитьТип())
    для ЭлементМассива из ПараметрМассив
        Консоль.Записать("Индекс = " + ПараметрМассив.Найти(ЭлементМассива)
 + ", значение = " + ЭлементМассива)
    ;
;
```

В данном примере в качестве параметра метода `МетодСМассивом()` выступает значение типа `ФиксированныйМассив`. Это позволяет нам передавать в этот метод как обычный массив (1 и 3

случай вызова в методе Скрипт()), так и фиксированный массив (2 случай вызова). Такая возможность предоставляется потому, что тип Массив является потомком типа ФиксированныйМассив.

Если изменить типа параметра ПараметрМассив метода МетодСМассивом() на Массив, то попытка вызвать метод с фиксированным массивом в качестве параметра приведет к ошибке компиляции из-за несовпадения типов.

Сравнение массивов

Допустимо сравнение массивов на равенство (или неравенство). Два массива являются равными, если равны их размеры и каждый элемент первого массива, содержится во втором массиве. Если элемент массива является другим массивом - выполняется попытка рекурсивного сравнения. следует использовать тип сравнивать значения типа Массив и ФиксированныйМассив.

Множество

Описывает работу с коллекцией, которая может содержать только уникальные элементы.

Множество - это коллекция, которая содержит **только уникальные** элементы. Порядок обхода элементов соответствует порядку добавления элементов в множество. Множество позволяет хранить элементы произвольных типов.

Множество могут быть неизменяемыми и изменяемыми. Как следует из самого термина, в неизменяемом множестве нельзя изменить добавлять или удалять значения множества. Наряду с термином "неизменяемый" может применяться термин "фиксированный". В то же время, изменяемое множество - это множество, для которого поддерживается изменение списка элементов множества. Изменяемое множество будет называться множеством или обычным множеством. Соответственно, тип для неизменяемого множества называется ФиксированноеМножество, а тип для обычного множества - Множество.

Множество может быть описано пустым и с указанием данных инициализации. Данные инициализации должны располагаться в фигурных скобках "{}". В данных инициализации допускается указывать повторяющиеся значения. Они будут отброшены без формирования ошибок.

```
метод Скрипт()
    пер ПустоеМножество: Множество
    пер МножествоСДанными = {1, 2, 3}
;
```

При выполнении проверки вхождения, метод СодержитВсе() проверяет полное вхождение, т.е. результат будет равен Истина в том случае, если в исходном множестве содержаться все элементы из коллекции, которая является параметром метода СодержитВсе().

Важной особенностью множеств является возможность выполнять над ними специальные операции:

- Проверка на равенство. Два множества являются равными, если равны их размеры и каждый элемент первого множества, содержится во втором множестве. Допускается сравнивать значения типа Множество и ФиксированноеМножество.
- Объединение множеств (метод Объединение()). В этом случае формируется множество, которое содержит все уникальные значения двух исходных множеств.
- Пересечение множеств (метод Пересечение()). В этом случае формируется множество, которое содержит только те элементы, которые присутствуют одновременно в обоих множествах.
- Разность множеств (методы Разность() и СимметрическаяРазность()). Простая разность формирует результирующее множество, которое содержит только те элементы исходного множества, которые отсутствуют в множестве, которое является параметром метода. Симметрическая разность формирует результирующее множество, которое содержит только те элементы, которые уникальны в каждом множестве. Другими словами, результат работы метода СимметрическаяРазность() не будет включать в себя значения, которые есть одновременно в обоих множествах.

```
метод Скрипт()
    пер Множество1 = {1, 2, 3, 4}
```

```
    пер Множество2 = {3, 4, 5, 6}
    Консоль.ЗаписатьСтроковоеПредставление("Множество1  : " + Множество1)
    Консоль.ЗаписатьСтроковоеПредставление("Множество2  : " + Множество2)
    Консоль.ЗаписатьСтроковоеПредставление("Объединение : "
 + Множество1.Объединение(Множество2))
    Консоль.ЗаписатьСтроковоеПредставление("Пересечение : "
 + Множество1.Пересечение(Множество2))
    Консоль.ЗаписатьСтроковоеПредставление("Разность    : "
 + Множество1.Разность(Множество2))
    Консоль.ЗаписатьСтроковоеПредставление("Сим.разность: "
 + Множество1.СимметрическаяРазность(Множество2))
;
```

Доступ к элементам множества поддерживается только с помощью цикла для в варианте обхода коллекции. Доступ к элементам множества с помощью функционального или операторного способов не поддерживается.

В цикле обхода коллекции не поддерживается изменение состава коллекции. При попытке такого изменения будет выброшено исключение.

```
метод Скрипт()
    ВывестиМножество({1, "2", 3, Истина, [1, 2, 3]})
;

метод ВывестиМножество(ЧтоВыводим: ФиксированноеМножество)
    Консоль.ЗаписатьСтроковоеПредставление("Тип параметра: "
 + ЧтоВыводим.ПолучитьТип())
    Консоль.ЗаписатьСтроковоеПредставление("Размер параметра: "
 + ЧтоВыводим.Размер())
    для ЭлементКоллекции из ЧтоВыводим
        Консоль.Записать("Значение = " + ЭлементКоллекции + ", тип значения
 = " + ЭлементКоллекции.ПолучитьТип())
    ;
;
```

Вывод типа значения множества в данном примере призван показать, что одно множество может содержать значения разного типа.

Соответствие

Описывает коллекцию значений, каждое из которых имеет уникальный идентификатор (а рамках коллекции).

Общее описание

Соответствие - это коллекция объектов типа КлючИЗначение. Порядок обхода элементов соответствует порядку добавления элементов в соответствие. Соответствие позволяет хранить элементы произвольных типов.

Соответствие могут быть неизменяемыми и изменяемыми. Как следует из самого термина, в неизменяемом соответствии нельзя изменять элементы добавлять или удалять значения множества. Наряду с термином "неизменяемый" может применяться термин "фиксированный". В то же время, изменяемое множество - это множество, для которого поддерживается изменение списка элементов множества. Изменяемое множество будет называться множеством или обычным множеством. Соответственно, тип для неизменяемого множества называется ФиксированноеСоответствие, а тип для обычного множества - Соответствие.

Объект **КлючИЗначение**

Как уже было сказано выше, соответствие является коллекцией элементов типа КлючИЗначение. Значение данного типа является объектом, который предоставляет два свойства: Ключ и Значение. В качестве некоторой аналогии можно представить ситуацию следующим образом: Ключ - это имя переменной, которой присваивается Значение. Однако аналогия является достаточно условной, т.к. в качестве имени

переменной может выступать только строка символов определенного формата, а в качестве значения свойства КлючИЗначение.Ключ может выступать любой объект языка.

Создание **Соответствия** и заполнение значениями

Соответствие можно создать несколькими способами:

- С помощью конструктора пустого соответствия.
- С помощью данных инициализации.

Данные инициализации заключаются в символы "{}". Сами данные инициализации состоят из пар "ключ-значение". Пары разделяются символом ",". Значение ключа отделяется от собственно значения символом ":".

```
метод Скрипт()
    // пустое соответствие
    пер КурсыВалют1 = {:}
    // создаем заполненное соответствие с данными инициализации
    пер КурсыВалют2 = {"Рубль РФ": 1, "Рубль Белоруссии": 31.01}
    // так тоже можно сделать
    пер КурсыВалют3: Соответствие
    КурсыВалют3 = новый Соответствие()
    КурсыВалют3.Вставить("Рубль РФ", 1)
    КурсыВалют3["Рубль Белоруссии"] = 31.01
    // можно добавить значения, если нельзя это сделать сразу
    КурсыВалют1.Вставить("Армянский драм", 0.14)
    КурсыВалют1.Вставить(новый КлючИЗначение("Казахстанский тенге", 0.17))
;
```

Если на момент создания соответствия неясно, какие данные в нем будут храниться, то добавить эти данные можно потом. В примере выше так делается с соответствиями КурсыВалют1 и КурсыВалют3.

Если к моменту вставки элемента соответствие уже содержит значение с указанным ключом, то значение в соответствии будет заменено на значение, которое вставляется в рассматриваемом вызове. Другими словами, в результате исполнения следующего кода, в качестве значения элемента соответствия с ключом пи будет значение 3.141592654, а не просто 3:

```
метод Скрипт()
    пер Значения: Соответствие
    Значения.Вставить("пи", 3)
    Значения.Вставить("пи", 3.141592654)
;
```

Если необходимо создать копию какого-либо соответствия, то это можно сделать с помощью конструктора копирования. В этом случае новое соответствие будет являться полной копией исходного.

Соответствие, также, позволяет вставить какую-либо пару значений только в том случае, если этой пары еще нет в соответствии. Метод ВставитьЕслиОтсутствует() создаст новый элемент в соответствии в том случае, если ключ, используемый в методе, отсутствует в соответствии. Если ключ в соответствии уже есть, то метод ВставитьЕслиОтсутствует() не будет изменять существующую пару. При выполнении проверки анализируется только значение ключа.

Очистка соответствия выполняется либо методом Очистить() - при этом из соответствия удаляются все элементы, или с помощью метода Удалить(), который удаляет элемент к указанным ключом.

Элементы соответствия и работа с ними

Обращение к элементу соответствия может быть выполнено двумя способами:

- Операторный способ. В этом случае в квадратных скобках указывается ключ, значение которого необходимо получить. Например, выражение Значения["пи"] вернет значение, сохраненное в

соответствии Значения с ключом пи. Если в данном соответствии не существует элемента с ключом пи, то будет вызвано исключение. Операторный способ можно использовать как в правой, так и в левой части оператора присваивания. В случае использования в левой части оператора присваивания, такой способ работает аналогично методу Вставить().

- Функциональный способ. В этом случае нужно использовать метод Получить(), в качестве параметра которого выступает значение ключа. Обращение функциональным способом будет выглядеть следующим образом: Значения.Получить("пи"). Кроме метода Получить() можно использовать метод ПолучитьИлиУмолчание(). Этот метод вернет значение, соответствующее указному ключу, если ключ имеется в соответствии, и значение по умолчанию, если ключ отсутствует в соответствии. Значение по умолчанию указывается в качестве параметра метода ПолучитьИлиУмолчание().

Кроме получения значения соответствия по индексу, предоставляется возможность обойти коллекция с помощью операторов для (в обеих вариантах).

При использовании соответствия частым вариантом использования является проверка наличия в соответствии какого-либо ключа или значения. Выполнить такую проверку можно с помощью методов СодержитКлюч() и СодержитЗначение(). Также соответствие позволяет получить фиксированную коллекцию, которая содержит ключи или значения исходного множества. Для выполнения этого действия предназначены методы Ключи() и Значения(). Как следует из типа получаемого значения, полученные множества доступны только на чтение.

Сравнение соответствий

Допустимо сравнение соответствий на равенство (или неравенство). Два соответствия будут считаться равными, если размеры этих соответствий совпадают, ключи одного соответствия являются ключами другого, а также попарно совпадают все значения одинаковых ключей. Допускается сравнивать значения типа Соответствие и ФиксированноеСоответствие.

Перечисление

Описывает, что такое перечисление и как с ним работать.

Во время разработки какой-либо программы часто возникает необходимость использования некоторого набора сущностей, объединенных одним смыслом. Например, разрабатывая программу, которая может работать под управлением нескольких операционных систем, мы можем захотеть идентифицировать каждую из них. Можно использовать для такой идентификации строковые значения, но лучше для этих целей использовать специальный тип данных - *перечисление*. Перечисление - это специальный тип данных, множество значений которого представляет собой явно заданный набор идентификаторов. После объявления собственного перечисления, разработчик может использовать его наряду с перечислениями, которые уже существуют в языке. Для обращения к элементу перечисления следует указать имя самого перечисления и, через точку, требуемый элемент перечисления.

Синтаксис объявления перечисления следующий:

```
перечисление <ИмяПеречисления>
    Элемент1 [умолчание,]
    [Элемент2,]
    [Элемент3,]
    ...
;
```

ИмяПеречисления — ИмяПеречисления указывает имя создаваемого перечисления. Это имя в дальнейшем следует указывать в качестве имени типа при объявлении переменных, принимающих значения данного перечисления.

ЭлементХ

Имена, указанные в синтаксисе как ЭлементХ, задают тот самый набор уникальных идентификаторов (в рамках этого перечисления), который требуется разработчику. Например, для задания перечисления ОперационныеСистемы можно использовать следующий пример:

```
перечисление ОперационныеСистемы
    Windows,
    macOS,
    Linux
;
```

В этом примере значения Windows, macOS и Linux - это и есть тот самый набор уникальных идентификаторов.

В составе перечисления должно быть минимум одно значение.

умолчание

Перечисление может обладать единственным значением по умолчанию или вовсе не иметь значения по умолчанию. Значение по умолчанию будет использоваться в том случае, если переменная описывается с типом создаваемого перечисления и для этой переменной не указывается значение инициализации. Значением по умолчанию может быть любой элемент перечисления.

При использовании значения перечисления необходимо указать имя перечисления и конкретное значение перечисления. Разделителем этих идентификаторов случит символ ".". Таким образом, если в тексте программы необходимо использовать ссылку на операционную систему Windows, то это будет нужно сделать следующим образом: ОперационныеСистемы.Windows (пример такого перечисления приводится ранее в этом разделе).

В тоже время, значение перечисления без упоминания имени самого перечисления (имени типа) допускается в кратком виде оператора выбор с выбором из значений перечисления. Пример такого использования приводится ниже.

Использование перечисления

```
перечисление ОперационныеСистемы
    Linux,
    macOS,
    Windows умолчание
;

метод Скрипт()
    пер ИспользуемаяОС: ОперационныеСистемы
    ИспользуемаяОС = ОперационныеСистемы.Windows
    Консоль.Записать(Строка(ИспользуемаяОС))
;
```

Использование перечисления в операторе выбора

```
перечисление ОперационныеСистема
    Linux,
    macOS,
    Windows умолчание
;
```

```
метод Скрипт()
    пер ИспользуемаяОС: ОперационныеСистема

    выбор ИспользуемаяОС
    когда Windows
        // делаем что-то в случае Windows
    когда Linux, macOS
        // делаем что-то, если macOS или Linux
    ;
;
```

Структура

Содержит описание работы с объектом структура.

Общее описание

Структура представляет из себя тип, имеющий фиксированный набор полей и конструкторов. Каждое поле имеет тип, указанный при разработке структуры. Структура может содержать только данные и не может содержать описания методов. После описания структуры, ее можно использовать в программах наряду с другими типами данных.

Синтаксис объявления структуры:

```
структура <ИмяСтруктуры>
    [модификатор ИмяПоля1: ТипПоля1[ = ЗначениеИнициализации1]]
    [модификатор ИмяПоля2: ТипПоля2[ = ЗначениеИнициализации2]]
    [модификатор ИмяПоля3: ТипПоля3[ = ЗначениеИнициализации3]]
    ...
    [конструктор()]
    [конструктор(ИмяПоля1)]
    ...
;
```

ИмяСтруктуры

ИмяСтруктуры указывает имя описываемой структуры (имя типа). С помощью этого имени в дальнейшем будет происходить использование создаваемой структуры, например, создание переменной с типом структуры. Имя должно удовлетворять общим требованиям к именам в языке.

модификатор

Каждое поле структуры объявляется аналогично обычной переменной в модуле. Оно начинается с ключевого слова пер или знч, затем следует имя поля, его тип и необязательное значение инициализации. Имена полей должны соответствовать общим требованиям к именам языка.

: Поле структуры не может быть описано таким образом, чтобы у него был тип самой определяемой структуры. Другими словами не допускается рекурсивное объявление типа поля.

Имена полей должны быть уникальны в рамках одной структуры.

Если описание поля структуры начинается с модификатора пер, то значение такого поля может быть изменено путем присваивания значения. Если

	описание поля структуры начинается с модификатора `знч`, то для такого поля значение может быть установлено только во время создания экземпляра структуры с помощью конструктора.
ЗначениеИнициализации	Описывает, какое значение будет принимать поле структуры при создании объекта. В качестве значения инициализации может выступать любое выражение, которое вычислимо во время компиляции.
конструктор	По умолчанию, для структуры создается автоматический конструктор, который включает в себя все поля структуры в том порядке, как они описаны в самой структуре. Конструктор задается с помощью указания ключевого слова `конструктор`, за которым, в круглых скобках, должны быть указаны имена полей, инициализация которых будет выполняться этим конструктором. Во всех конструкторах должны быть перечислены поля структуры, типы которых не обладают значениями по умолчанию и для которых не указаны данные инициализации. Для параметров конструктора не допускается указание значений по умолчанию. Порядок полей в конструкторе может не совпадать с порядком полей в описании структуры. Структура может иметь несколько конструкторов, которые различаются только количеством параметров. Если при использовании структуры используется автоматический конструктор, то не рекомендуется менять порядок полей в объявлении структуры. Это приведет к изменению параметров автоматического конструктора и, как следствие, к появлению различных побочных эффектов. Например, перестанут компилироваться модули из-за несовпадения типов.

Работа со структурой

Очевидно, что первым шагом к использованию собственного типа данных является описание этого типа. Рассмотрим простой пример данных, которые может быть востребован в языке: сохранить информацию о каком-либо файле. Допустим, нам необходимо хранить о файле следующую информацию:

- Имя файла.
- Тип файла: исполняемый или не исполняемый.
- Полный путь к файлу.
- Размер файла.

Для этого создадим следующую структуру:

```
структура ОписаниеФайла
    пер ИмяФайла: Строка = "" // имя файла
    знч Исполняемый: Булево = Ложь // тип файла
    пер ПолныйПуть: Строка = "" // полный путь
    пер РазмерФайла: Число = 0 // размер файла
;
```

Если потребуется описать переменную с типом `ОписаниеФайла`, то сделать это можно обычным образом: `пер Файл: ОписаниеФайла;`. Если надо сразу инициализировать переменную некоторым значением, то это можно сделать следующим образом (на примере командного процессора ОС Windows 10 версии 1903):

```
структура ОписаниеФайла
    пер ИмяФайла: Строка = "" // имя файла
    знч Исполняемый: Булево = Ложь // тип файла
    пер ПолныйПуть: Строка = "" // полный путь
    пер РазмерФайла: Число = 0 // размер файла
;

метод Скрипт()
    пер КомандныйПроцессор: ОписаниеФайла = новый ОписаниеФайла("cmd.exe",
            Истина,
            "C:\\Windows\\System32\\cmd.exe",
            280064)
;
```

В данном примере видно, каким образом формируются данные инициализации. Данными инициализации выступают значения, указанные через запятую. В примере у структуры явно не указано ни одного конструктора, поэтому используется конструктор по умолчанию. Следовательно, значения, указанные в конструкторе, будут присваиваться полям структуры в следующем порядке: значения конструктора слева направо будут присваиваться полям структуры сверху вниз:

- Свойству `ИмяФайла` будет присвоена строка `cmd.exe`.
- Свойству `Исполняемый` будет присвоена значение `ложь`.
- Свойству `ПолныйПуть` будет присвоена строка `C:\\Windows\\System32\\cmd.exe`.
- Свойству `РазмерФайла` будет присвоена значение 280 064.

Стоит обратить внимание, на то, что свойство `Исполняемый` указано с модификатором `знч`. А это означает, что написать выражение `КомандныйПроцессор.Исполняемый = ложь` уже не получится.

Очевидно, что в реальной жизни параметры файла следует заполнять, предварительно получив эти значения у файловой системы, но в примере можно пойти некоторое упрощение и заранее получить нужные значения.

Если теперь нам необходимо получить значение какого-то поля структуры, то для этого потребуется указать имя переменной и имя поля, разделив их символом "точка", вот так: `КомандныйПроцессор.ИмяФайла` - вернет значение, которое находится в поле `ИмяФайла` переменной `КомандныйПроцессор`, которая описана с типом структуры `ОписаниеФайла`. Указанное выражение может использоваться как в качестве источника - для получения значения поля, так и в качестве назначения выражения - для установки значения поля.

Кроме указанного способа можно использовать операторный способ получения значения поля структуры. Для этого следует в квадратных скобках после имени переменной указать имя свойства, с которым надо работать: `КомандныйПроцессор["ИмяФайла"]` или так: `КомандныйПроцессор[стрИмяПоля]`.

Сравнение структур

Допустимо сравнение двух значений типа структура на равенство (или неравенство). Две структуры будут считаться равными, если эти структуры одного типа и значения полей попарно совпадают.

Понятия, связанные с данным

Переменные и работа с ними

В языке все имеет свое *имя*. Имена используются для переменных, объявляемых пользователем. С помощью имен выполняется идентификация объектов и их свойств и т.д. В данном разделе будут рассмотрены правила формирования имен, правила объявления переменных, доступ к данным, находящимися в переменных, а также особенности, связанные с видимостью имен и доступностью значений.

Операторы и управляющие конструкции

Содержит описание управляющих конструкций, которые позволяют разработчику изменять последовательный порядок исполнения инструкций языка.

Условные операторы

Приводится описание условных операторов.

Условный оператор **если**

Язык предоставляет возможность выполнять разные фрагменты кода в зависимости от результата логического выражения. Условный оператор имеет следующий синтаксис:

```
если <ЛогическоеВыражение>
    // операторы языка
[иначе [если <ЛогическоеВыражение>]
    // операторы языка]
;
```

Условный оператор работает по простой схеме. В качестве условия может выступать только логическое выражение. Результатом вычисления логического выражение может быть только два значение: Истина и Ложь. Следовательно, если результатом вычисления условия будет значение Истина, то будет выполнен программный код, который идет на строке, следующей за строкой с условием. Если результатом вычисления условия будет значение Ложь, то будет выполнен программный код между ключевым словом иначе и окончанием блока (символ ";"). Блок иначе является необязательным элементом оператора если. Это означает, что условный оператор должен обязательно включать только тот код, который должен быть выполнен в том случае, если результатом вычисления логического выражения стало значение Истина.

В теле условного оператора допускаются любые операторы языка. Это означает, что внутри одного условного оператора может располагаться произвольное количество вложенных условных операторов, операторов цикла и т.д. При создании вложенных конструкций следует помнить, что в языке отсутствует оператор безусловного перехода, поэтому быстрый выход из глубоких вложенных условий может быть непростой задачей.

Примеры использования условного оператора

```
метод Скрипт()
    пер А = 0

    // минимальный условный оператор
    если А == 5

 // здесь окажемся только в том случае, когда значение переменной А будет равно
 5
    ;

    // другой вариант условного оператора
    если 1 > 2
        // Если попадем сюда - будет
    иначе
        // но всегда будем попадать сюда, т.к. чудес не бывает
    ;

    // так тоже можно писать
    если А == 0
        Консоль.Записать("А равно 0")
    иначе если А < 10
        Консоль.Записать("А меньше 10")
    иначе если А > 10 и А < 15
        Консоль.Записать("А больше 10 но меньше 15")
```

```
    иначе
        Консоль.Записать("А строго больше 15")
    ;
;
```

Условный оператор **выбор**

Оператор выбор является альтернативой оператору если. Оператор может использоваться в двух формах: полной и предикатной. Предикатная форма отличается от полной отсутствием выражения выбора.

```
выбор [<ВыражениеВыбора>]
когда <ВыражениеСравнения>[, <ВыражениеСравнения>]
    // операторы языка
[когда <ВыражениеСравнения>[, <ВыражениеСравнения>]
    // операторы языка]
[иначе
    // операторы языка]
;
```

ВыражениеВыбора — Это выражение, которое будет находиться в левой части логического выражения при проверке условий.

ВыражениеСравнения — Это выражение, которое будет находиться в правой части логического выражения при проверке условий. В зависимости от формы оператора выбор, ВыражениеСравнения может иметь несколько видов:

- Полноценное выражение. В этом случае будет проверяться следующее условие: ВыражениеВыбора == ВыражениеСравнения. Если используется предикатная форма оператора, то ВыражениеСравнения должно представлять из себя полноценное логическое выражение, которое возвращает результат Истина или Ложь.
- Частичное бинарное выражение: <ЛогическийОператор> <ВыражениеСравнения>. В такое выражение могут входить операторы сравнения (==, != и т.д.) и оператор проверки типа это. В этом случае будет проверяться следующее условие: ВыражениеВыбора <ОператорСравнения> <ВыражениеСравнения>.

Если выражения сравнения перечислены через символ "," (запятая), то это означает, что перечисленные через запятую условия будут объединены "по ИЛИ".

Оператор работает следующим образом:

1. Полная форма
   - Выполняется вычисление значения ВыражениеВыбора.
   - Для каждого условия (или набора условий), перечисленных в ветках когда выполняется сравнение ВыраженияВыбора и ВыраженияСравнения.
   - Если результат сравнения равен значению Истина, то выполняются инструкции языка после соответствующей строки и затем управление передается на инструкцию языка, следующую за

символом ";", закрывающим оператор выбор. Проверка условий выполняется в порядке следования операторов когда.

2. Предикатная форма
   - Для каждой ветки когда выполняется вычисление выражения ВыражениеСравнения.
   - Если результат сравнения равен значению Истина, то выполняются инструкции языка после соответствующей строки и затем управление передается на инструкцию языка, следующую за символом ";", закрывающим оператор выбор. Проверка условий выполняется в порядке следования операторов когда.

При любом варианте использования оператора выбор, если ни одно из условий не было выбрано, а в операторе присутствует ветвь иначе - управление будет передано этому набору инструкций.

Оператор выбора

```
метод Скрипт()
    пер Числа = [0, 1, 2, 3, 4, 5, 6, 7]
    для Значение из Числа
        выбор Значение
        когда 1
            Консоль.Записать("1")
        когда 2, 3
            Консоль.Записать("2 или 3")
        когда > 4
            Консоль.Записать("больше 4")
        иначе
            Консоль.Записать("все остальное: " + Значение)
        ;
    ;
;
```

Тернарный оператор **?**

Кроме операторов если и выбор, существует еще один оператор, который позволяет выполнить некоторое действие в зависимости от условия. Это тернарный (троичный) оператор ?. В отличие от оператора если, оператор ? не выполняется передачу управления, а возвращает один из своих операндов (второй или третий), в зависимости от значения логического выражения, заданного первым операндом. Из этого описания становится очевидным синтаксис оператора:

```
<Логическое выражение> ? <Выражение Истина> : <Выражение Ложь>
```

Схема работы оператора выглядит достаточно просто:

1. Вычисляется логическое выражение, заданное первым оператором. Результатом вычисления логического выражение может быть только два значение: Истина и Ложь. В зависимости от результата вычисления логического выражения, происходит вычисление только одного из двух выражений: Выражение Истина или Выражение Ложь.
2. Если результат вычисления логического выражение равен Истина, то тернарный оператор возвращает результат вычисления Выражение Истина.
3. Если результат вычисления логического выражение равен Ложь, то тернарный оператор возвращает результат вычисления Выражение Ложь.

Применение тернарного оператора оправдано в том случае, когда необходимо сделать бинарный выбор: или одно значение (или выражение) или другое.

Примеры использования тернарного оператора

```
метод Скрипт()
    пер Возраст = 10
```

```
    пер ОписаниеДокумента = Возраст >= 14 ? "Нужен паспорт" :
 "Нужно свидетельство о рождении"
;
```

Понятия, связанные с данным

Обращение к переменной
Описывает различные варианты обращения к значению переменной.

Циклы

В разделе приводится описание операторов организации циклов.

Циклы предназначены для многократного повторения одного и того же фрагмента кода. Этот фрагмент кода будет называться *телом цикла*. Одно повторение тела цикла будет называться *итерацией*. В языке существует несколько вариантов организации циклов.

Цикл **для** (со счетчиком)

Цикл для имеет следующий синтаксис:

```
для <СчетчикЦикла> = <Выражение1> по <Выражение2>
    // тело цикла
    [прервать]
    // тело цикла
    [продолжить]
    // тело цикла
;
```

В этом описании:

СчетчикЦикла

Данная переменная управляет выполнением итераций. Эта переменная так и называется - *счетчик цикла*. Данная переменная имеет тип Число и "объявляется" с модификатором знч. Цикл будет выполняться до тех пор, пока переменная СчетчикЦикла принимает значения между результатом вычисления Выражение1 и Выражение2. Переменная, выступающая в виде счетчика цикла, не может быть объявлена вне цикла. Областью видимости счетчика цикла является тело цикла. Это открывает возможность использовать одно и тоже имя счетчика цикла в разных циклах в одном блоке кода (не вложенных!), не опасаясь различных сторонних эффектов.

Цикл будет продолжаться до тех пор, пока счетчик цикла меньше или равен условию завершения цикла, которое определяется Выражение2. Система автоматически увеличивает значение счетчика цикла на 1 после окончания каждой итерации.

Выражение1

Результат вычисления данного выражения будет присвоен переменной СчетчикЦикла и это значение будет использоваться при первой итерации. Исключением является случай, когда начальное значение счетчика цикла будет строго больше, чем значение Выражение2, которое определяет условие завершения цикла. В этом случае цикл сразу завершится, без выполнения итераций.

Выражение2	Выражение, определяющее условие завершения цикла.
продолжить	В том случае, когда необходимо досрочно завершить выполнение очередной итерации цикла и перейти к следующему значению счетчика цикла, можно использовать ключевое слово продолжить. В результате итерация прервется в том месте, где указано это ключевое слово, счетчик цикла будет увеличен и итерация начнется с самого начала (если цикл не будет завершен).
прервать	Данное ключевое слово используется в том случае, когда надо завершить цикл "прямо сейчас", не дожидаясь штатного завершения цикла (по значению счетчика цикла).

Таким образом, цикл для работает следующим образом:

1. Счетчику цикла (переменная СчетчикЦикла) присваивается начальное значение. Начальное значение получается путем вычисления выражения Выражение1.
2. Проверяется необходимость завершения цикла. Для этого вычисляется Выражение2 и проверяется, что значение переменной СчетчикЦикла меньше или равно результату вычисления выражения. Если значение СчетчикЦикла больше значения Выражение2 - цикл завершается. Вычисление значения Выражение2 происходит один раз, в начале цикла, и не пересчитывается при каждой итерации.
3. Выполняется тело цикла.
   - Если во время выполнения тела цикла обнаруживается ключевое слово продолжить, то значение счетчика цикла увеличивается на 1 и выполнение продолжается с шага 2.
   - Если во время выполнения тела цикла обнаруживается ключевое слово прервать, то выполнение тела цикла прерывается безусловно и управление передается на оператор, который следует за последним оператором цикла.
4. Значение счетчика цикла увеличивается на 1 и выполнение продолжается с шага 2.

Цикл **для (**счетчик**)**

```
метод Скрипт()
    пер Сумма = 0
    для Индекс = 1 по 50
        Сумма += Индекс
    ;
    Консоль.Записать("Результат = " + Сумма)
;
```

Цикл **для (**обход коллекции**)**

Если необходимо перебрать элементы какой-либо коллекции, то можно использовать специальный вариант цикла, который предназначен для обхода коллекций. С помощью такого цикла можно обходить потомки базового типа Обходимое и значения типа Строка. Рассмотрим этот вариант цикла более подробно.

Вариант цикла для, использующегося для обхода коллекций, имеет следующий синтаксис:

```
для <ЗначениеЭлемента> из <Коллекция>
    // тело цикла
    [прервать]
    // тело цикла
    [продолжить]
    // тело цикла
;
```

Существенная разница от "обычного" цикла для заключается в следующем:

- Нет переменной, выступающей в роли счетчика цикла. Вместо счетчика цикла существует переменная ЗначениеЭлемента, в которую система помещается ссылку на очередной элемент коллекции. Переменная "объявляется" с модификатором знч, что исключает присваивание этой переменной какого-либо значения из встроенного языка. С помощью этой переменной разработчик получает доступ к значению элемента коллекции и может выполнять все те действия, которые этот элемент поддерживает.
- Отсутствует выражение, определяющее условие завершения цикла. При обходе коллекции цикл будет завершен тогда, когда будет перебрана вся коллекция.

В остальном все просто:

ЗначениеЭлемента	В данную переменную помещается значение очередного элемента коллекции. Это значение изменяется после окончания каждой итерации цикла. Во время первой итерации данная переменная указывает на первый элемент коллекции. Если для коллекции не определен порядок обхода элементов, то при каждом обходе коллекции может наблюдаться другой порядок элементов.
Коллекция	Коллекция, которую необходимо обработать.

Принудительное завершение текущей итерации (с помощью ключевого слова продолжить) и принудительное завершение цикла (с помощью ключевого слова прервать) работают аналогично циклу для со счетчиком цикла.

Цикл **для (**обход коллекции**)**

```
метод Скрипт()
    для Элемент из [1, 3, 5, 7, 9]
        Консоль.Записать("Элемент набора - " + Элемент)
    ;
;
```

Цикл **пока**

Цикл пока является еще один вариантом организации циклов. Данный вариант цикла будет выполнять тело цикла до тех пор, пока условие цикла истинно. Это самый универсальный вариант организации цикла, т.к. не накладывается никаких условий на то, каким образом будет организован счетчик цикла, каким образом будет организовано завершение итераций и какой объект (или объекты) будут обрабатываться в цикле.

Цикл пока имеет следующий синтаксис:

```
пока <Выражение>
    // тело цикла
    [прервать]
    // тело цикла
    [продолжить]
    // тело цикла
;
```

Как уже было сказано выше, цикл пока будет выполняться до тех пор, пока выражение Выражение при своем вычислении возвращает значение Истина. Как только в результате вычисления выражения будет получено значение Ложь - исполнение цикла пока будет прервано.

Принудительное завершение текущей итерации (с помощью ключевого слова продолжить) и принудительное завершение цикла (с помощью ключевого слова прервать) работают аналогично циклу для.

Цикл **пока**

```
метод Скрипт()

 // Цикл завершит свою работу, когда значение переменной Счетчик станет равно
 5
    пер Счетчик = 1
    пока Счетчик < 5
        Счетчик += 1
    ;

    // Пример бесконечного цикла
    пока Истина
        // тело цикла
    ;
;
```

Понятия, связанные с данным

Обращение к переменной
Описывает различные варианты обращения к значению переменной.

Методы

Фрагмент программного кода, к которому можно обратиться из другого места программы называется *методом*. Данный раздел описывает особенности описания и использования методов.

Общая информация

При разработке программы часто возникает необходимость повторного использования какого-либо фрагмента программы. Если нам нужно просто повторить этот фрагмент несколько раз - можно использовать цикл. А если этот фрагмент необходимо вызывать из разных мест программы - необходимо каким-то образом выделить программный код и оформить его специальным образом. Такой обособленный фрагмент кода называется *метод*. Метод всегда возвращает какое-либо значение. Подробнее про возвращаемое значение метода будет написано далее в этом разделе.

Для упрощения описания в данном документе могут использоваться два "сокращенных" упоминания метода с точки зрения возвращаемого значения:

- Если у метода явно указан тип возвращаемого значения, то для такого метода возможно использование термина *функция*.
- Если у метода явно не указан тип возвращаемого значения, то возможно использование термина *процедура*.

Подробнее о возвращаемом значении метода будет сказано далее.

Используемые определения

В процессе описания методов будут использоваться несколько терминов. Они уже могли использоваться и ранее в данном документе, но сейчас мы дадим им конкретные определения.

Термин	Определение
Определение метода	Место в исходном коде, где создан метод.
Заголовок метода	Часть определения метода, которая содержит ключевое слово `метод`, имя метода, описание формальных параметров в круглых скобках и комментарий, описывающий назначение метода.

Термин	Определение
Тело метода	Инструкции, расположенные между заголовком метода и символом `;`, описывающим завершение метода.
Вызов метода	Место в тексте программы, где указывается имя метода с указанием фактических параметров и использованием результата работы (только в случае возвращаемого значения).
Определение параметров и передача аргументов	*Параметры* (*формальные параметры*) - это имена переменных, через которые тело метода получает доступ к значениям, которые предоставляются методу во время его вызова. *Аргументы* (*фактические параметры*) - это значения, которые указываются во время вызова метода. Сопоставление формального параметра и фактического параметра выполняется по порядку следования значения в вызове метода (позиционное сопоставление). Предоставление аргументов методу при его вызове называется *передача параметров*.
Возврат значения	Передача вызывающему программному коду какого-либо значения при завершении работы метода. Метод не обязательно должен возвращать какое-либо значение.

Определение метода

Прежде чем использовать метод, его надо определить. Определение метода выглядит следующим образом:

```
экспорт
метод ИмяМетода([ФормальныйПараметр1[, ФормальныйПараметр2[, ...]]])[: Тип]
    // тело
    метода [возврат[ Значение]]
;
```

Каждый формальный параметр описывается следующим образом:

```
ИмяФормальногоПараметра: ТипПараметра[ = ЗначениеПоУмолчанию]
```

`ИмяФормальногоПараметра` задает имя формального параметра в соответствии со стандартными правилами формирования имен. `ТипПараметра` указывает, соответственно, тип формального параметра (в том числе и составной). `ЗначениеПоУмолчанию` является необязательным элементом описания формального параметра и позволяет указать значение по умолчанию, т.е. значение, которое будет присвоено формальному параметру, если при вызове метода данный параметра пропущен. Не допускается пропускать указание значения при вызове метода, если для соответствующего формального параметра не указано значение по умолчанию. В качестве значения по умолчанию выступает любое выражение, которое может быть вычислено на этапе компиляции:

- Литералы (включая литералы коллекций) являются вычислимыми
- Обращения к константам модулей являются вычислимыми.
- Операции над вычислимыми являются вычислимыми, кроме следующих исключений:
  - Вызовы методов и обращения к свойствам.
  - Вызовы конструкторов.
  - Операции сложения значения типа `Строка` со значением другого типа.

Особенности передачи параметров

Переменные хранят ссылки на объекты. В методы передаются значения переменных. Таким образом можно в методе изменять состояние объекта, но нельзя изменить значение переменной, передаваемого в качестве аргумента метода.

```
                метод Скрипт()
    пер ДляТеста = 5
    Тестовая(ДляТеста)
    Консоль.Записать("ДляТеста = " + ДляТеста)
;

метод Тестовая(Параметр1: Число)
    Параметр1 = 6
;
```

В данном примере, значение переменной `ДляТеста` в методе `Скрипт()` будет равно 5 и до вызова процедуры `Тестовая()` и после вызова этой процедуры. Такое поведение обеспечивает передача параметра "по значению".

Однако, если в метод передается объект, который располагает методами для изменения своего содержимого, то ситуация будет немного другой. Рассмотрим пример:

```
                  метод Скрипт()
    пер ДляТеста = [1, 2]
    Тестовая(ДляТеста)
    Консоль.Записать("ДляТеста = " + ДляТеста)
;

метод Тестовая(Параметр1: Массив)
    Параметр1.Очистить()
;
```

В этом примере до выполнения процедуры `Тестовая()`, в массиве `ДляТеста` будет 2 значения: 1 и 2. После вызова процедуры `Тестовая()`, в массиве не будет ни одного элемента. В то же время, если в процедуре `Тестовая()` мы попробуем присвоить формальному параметру `Параметр1` какое-либо значение - это не приведет к каким-либо изменениям аргумента (переменной `ДляТеста`).

Завершение метода и возвращаемое значение

Метод завершает свою работу после того, как завершается исполнение тела метода. Другим способ прервать исполнение метода является указание в необходимом месте метода оператора `возврат`.

Если метод должен вернуть какое-либо значение, необходимо это значение указать в качестве параметра оператора `возврат`. При этом тип фактически возвращаемого значения должен совпадать с типом возвращаемого значения, который указан при описании метода. Тип возвращаемого значения указывается после окончания списка формальных параметров метода. Тип указывается после символа ":" (двоеточие). Однако, если метод, в котором нет оператора `возврат` (или, другими словами, в котором не предполагается возвращаемого значения) будет использован в каком-то выражении или в качестве правого значения оператора присваивания, то в качестве возвращаемого значения такого метода будет выступать значение `Неопределено`.

Повторное определение (перегрузка) методов

При разработке может возникнуть необходимость создать метод, который отличается количеством или типом параметров. При этом выполняемое действие существенно зависит от типа или количества параметров. Такое

поведение можно реализовать двумя способами: составным типом параметров и значениями параметров по умолчанию или повторным определением метода. Повторное определение метода заключается в том, что вы можете описать несколько методов с одним именем. Метод, который имеет несколько заголовков с разным составом параметров будет называться *перегруженным*. При этом нужно помнить, что любой вызов перегруженного метода в исходном коде должен позволять компилятору **однозначно** определить, какой вариант метода необходимо использовать в данном случае. Если определить используемый вариант метода во время компиляции не представляется возможным, произойдет ошибка компиляции. Перегруженные функции не могут различаться только типом возвращаемого значения.

Рассмотрим примеры корректного и некорректного определения перегруженных методов:

```
1. метод Пример1(Параметр1: Строка|Число)
2. метод Пример1(Параметр1: Строка)
3. метод Пример1(Параметр1: Строка|Булево)

4. метод Пример2(Параметр1: Объект)
5. метод Пример2(Параметр1: Число)
6. метод Пример2(Параметр1: Число, Параметр2: любой)

7. метод Пример3(Параметр1: Строка)
8. метод Пример3(Параметр1: Число)
9. метод Пример3(Параметр1: Строка, Параметр2: любой = 22)

10. метод Пример4(Параметр1: Строка): Строка
11. метод Пример4(Параметр1: Число): Число
12. метод Пример4(Параметр1: Строка): Число
```

Процедуру `Пример1()` (строки 1, 2 и 3) нельзя переопределить предложенным способом. Текущее определение типов параметра процедуры не позволяет однозначно определить, какой вариант метода необходимо вызвать. Причина в том, что `Параметр1` описан с использованием составного типа, при этом в каждом варианте метода присутствует тип `Строка`.

Процедура `Пример2()` (строки 4, 5 и 6) можно описать только исключив строку 5. Причиной является тот факт, что тип `Объект` является базовым для всех типов системы. В то же время, заголовок процедуры в строке 6 имеет другое количество параметров.

Процедура `Пример3()` (строки 7, 8, 9) можно описать только исключив строку 9. Причиной является то, что второй параметр имеет значение по умолчанию, а следовательно указав один параметр (первый), компилятор не будет понимать, какой из вариантов метода использовать.

Наконец, функция `Пример4()` (строки 10, 11, 12) может быть описана или строками 10 и 11 или строками 11 и 12. Причиной является тот факт, что перегруженные функции не могут отличаться только типом возвращаемого значения. Поэтому вариант со строками 10 и 12 отпадает.

Также стоит отметить несколько особенностей, касающихся перегруженных методов:

- Основное назначение перегрузки метода - это удобство разработчика.
- Если описываемый алгоритм изначально ориентирован на то, что в качестве параметров передаются составные типы - не нужно выполнять перегрузку такого метода.
- Если алгоритм, описываемый методом, различается от типа переданного параметра, то стоит рассмотреть возможность перегрузки метода, реализующего такой алгоритм.
- Перегруженными могут быть не только методы, но и конструкторы.

Исключения

Данный раздел содержит описание работы с исключениями и закрываемыми ресурсами.

Общая информация

Во время работы любой программы могут возникать различные ошибки. Некоторые ошибки вызываются некорректными параметрами, некоторые ошибки возникают в результате каких-либо внешних факторов

(права доступа, доступность канала связи и т.д.). Некоторые ошибки позволяют каким-то образом продолжать работу метода или программы, какие-то ошибки не позволяют продолжить выполнение. 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, рекомендуется передавать в программный код, который инициировал выполнение прервавшегося действия. И уже вызывающий код будет принимать решение, что делать в случае возникновения той или иной ошибки.

Таким образом, мы понимаем, что информирование об ошибке состоит из двух частей:

1. Некоторого действия, которое укажет вызывающему коду, что обнаружена ошибка.
2. Набор данных, который описывает обнаруженную ошибку.

Для обнаружения ошибки служит механизм *исключений* и *обработки исключительных ситуаций* (или *обработка исключений*). Исключения (и механизм работы с ними) - это и есть то действие, которое сообщает об ошибке. А для получения информации об ошибке служат специальные *типы исключений*.

Исключение - это событие, которое возникает во время наступления нештатной ситуации (ошибки) при исполнении кода программы. Например, ошибка деления чисел на 0, ошибка доступа к файлу, выход индекса за границы массива и т.д. При наступлении ошибки будет *выброшено исключение*. Когда выбрасывается исключение, работа программы на текущем уровне вложенности прерывается и управление передается на предыдущий уровень по порядку вызова методов языка (по стеку вызовов). Если исключение не обработано на очередном уровне, то ошибка "поднимается" по уровням вложенности вызовов языка до тех пор, пока не достигнет собственно среды исполнения. Если исключение "дошло" до среды исполнения, то это означает следующее:

- Исполнение программы прервано.
- Все данные, которые были сформированы (и не сохранены) до возникновения ошибки - потеряны.
- Пользователь увидит техническое сообщение, которое может не нести никакого прикладного смысла для пользователя.

Чтобы упростить пользователю борьбу с последствиями ошибок, предназначен механизм *обработки исключений*. Это специальный механизм, который определяет наступление ошибки (выбрасывание исключения) и затем передает управление в специальное место программы, где можно выполнить действия, связанные с ошибкой. Такое место называется *обработчик исключений*. При этом информация об ошибке поступит в обработчик исключений в виде специального объекта языка (базовым типом для которого выступает `Исключение`), тип которого будем называть *типом исключения*.

Обработка исключений

Для обработки исключений предназначена специальная конструкция обработки исключений `попытка - поймать`:

```
попытка
    // безопасный код
[поймать ИмяПеременной1: ТипИсключения1
    // обработка исключения типа ТипИсключения1]
[поймать ИмяПеременной2: ТипИсключения2|ТипИсключения3
    // обработка исключений с типом ТипИсключения2 или ТипИсключения3]
[поймать ИмяПеременной3: Исключение
    // обработка исключений остальных типов]
[вконце
    // завершающая секция]
;
```

Рассмотрим, как работает описанная конструкция. Если в тексте модуля, отмеченного комментарием `// безопасный код` (причем на любом уровне вложенности вызовов от рассматриваемого места), произойдет ошибка, приводящая к выбрасыванию исключения, то:

1. Исполнение кода будет безусловно прервано. Станут недоступны те переменные, которые объявлены в той области видимости, где возникло исключение.

2. Будет определен тип исключения. В зависимости от типа исключения:
   - Подбор обработчика исключений происходит в порядке описания. Обработчик считается подходящим в том случае, если тип исключения обработчика совпадает с типом обрабатываемого исключению. При этом будет учитываться иерархия типов. Обработчиком исключения в данном случае называется блок кода, следующий после конструкции `поймать ИмяПеременной: ТипИсключения`.
   - В обработчик для типа `Исключение` управление будет передаваться для всех исключений, которые не были перехвачены ранее. Это произойдет потому, что тип `Исключение` является базовым типом для любого исключения.
   - Если в обработчике исключения нет обработки нужного типа исключения (включая обработчик любого исключения) - значит исключение будет передано на предыдущий уровень по стеку вызовов методов.
3. Перед тем, как управление будет передано коду, расположенному после синтаксического элемента `;`, завершающего блок `попытка - поймать`, управление передается блоку кода `вконце`. Сюда же управление предается и в том случае, если никакого исключения в безопасном коде не произошло. В данном блоке имеет смысл размещать код, который должен выполнить какие-то действия вне зависимости от того, удачно или неудачно завершился безопасный фрагмент кода. Блок `вконце` выполняется даже если в блоках `попытка` и `поймать` используются операторы `возврат`, `прервать` или `продолжить`. Тело блока `вконце` будет выполнено непосредственно перед возвратом значения или продолжением/прерыванием цикла. Хорошим примером может явится открытие файла в безопасном фрагменте и закрытие этого файла в блоке кода `вконце`. Очевидно, что закрывать файл в безопасном блоке кода в этом случае не требуется.

   Важное замечание: Секция `вконце` не может содержать управляющих конструкций, которые могут прервать исполнение секции (операторы `возврат`, `прервать`, `продолжить`).

Как вызвать исключение?

Исключения возникают по нескольким причинам:

- Выполнение операции, недопустимой в языке (например, деление на 0). В этом случае исключение будет сформировано системой 1С:Исполнитель.
- С помощью оператора `выбросить`.

Рекомендуемый способ выбрасывания исключения - это применение оператора `выбросить`. В качестве параметра данного оператора выступает объект, описывающий исключение. Тип такого объекта является потомком типа `Исключение`. Этот объект может быть создан с помощью конструктора или получен в обработчике какого-либо исключения. Можно выбрасывать исключения как для стандартных, так и для собственных типов исключений. Стандартное исключение можно создать самостоятельно, если для такого исключения в колонке `Конструктор` таблицы Таблица 7. Типы исключений языка установлена отметка `Да`.

Объявление типа исключения

Для того, чтобы создать новый тип исключения, следует воспользоваться специальной конструкцией, имеющей следующий синтаксис:

```
исключение <ИмяТипаИсключения>
    [пер ИмяПоля1: ТипПоля1]
    [пер ИмяПоля2: ТипПоля2]
    [знч ИмяПоля3: ТипПоля3]
    ...
    [конструктор(ИмяПоля1)]
    [конструктор(ИмяПоля1, ИмяПоля2)]
;
```

В некотором смысле, описание типа исключения подобно описанию структуры: каждый тип исключения может содержать поля, которые будут содержать информацию, специфическую для этого типа исключения. Следует помнить, что при описании исключения не допускаются значения инициализации. Кроме полей,

созданных разработчиком данного типа исключения, каждый тип исключения содержит некоторый набор предопределенных полей, описание которых приведено в разделе Описание исключения.

В описании исключения могут быть описаны конструкторы исключения. Они описываются с применением ключевого слова `конструктор`. Описание конструктора аналогично описанию конструктора для типа `структура`, но в качестве параметров, дополнительно к именам свойств, можно использовать поля `Описание` и `Причина` базового типа `Исключение`. Если описывается несколько конструкторов - они должны различаться числом аргументов. При объявлении конструктора не поддерживается указание значения по умолчанию и указание типов аргументов конструктора. Системой неявно генерируются следующие конструкторы (если явно не объявлены конструкторы с таким же количеством параметров):

- Поле `Описание` в качестве первого параметра и все поля исключения в порядке объявления - второй и последующие параметры.
- Поле `Описание` в качестве первого параметра, поле `Причина` в качестве последнего параметра и все поля исключения в порядке объявления - начиная со второго параметра и до поля `Причина`.

Рассмотрим пример пользовательского исключения. Допустим, нам необходимо выбросить исключение при ошибке чтения файла. Создадим для этого тип исключения `ИсключениеЧтенияФайла`, где будет дополнительное поле - имя файла.

```
исключение ИсключениеЧтенияФайла
    пер ИмяФайла: Строка
;
```

Теперь для того, чтобы создать объект, описывающий это исключения, следует написать следующий код: `новый ИсключениеЧтенияФайла("Ошибка чтения файла", "path\file.txt")`. Первым параметром конструктора идет текстовое представление ошибки, затем идут столько значений, сколько полей указано у нашего исключения и последним, необязательным параметром, является исключение, которое явилось причиной выбрасываемого исключения. В этом случае будет образовываться цепочка вложенных исключений. Наиболее вложенное исключение (или самое нижнее) будет описывать исходную причину ошибки, а самое верхнее исключение в цепочке будет содержать текст ошибки для пользователя. Когда необходимо сохранить исходную причину ошибки, но при этом все-таки сохранить сам факт исключения с тем, чтобы обработкой занимались обработчики исключения в вызывающем коде, следует создать свой тип исключения, а перехваченное исключение указать в качестве причины создаваемого. В случае рассматриваемого исключения `ИсключениеЧтенияФайла`, создание исключения будет выглядеть следующим образом: `новый ИсключениеЧтенияФайла("Ошибка чтения файла", "path\file.txt", ФайловаяОшибка)`. В этом примере `ФайловаяОшибка` - это объект, описывающий исключение, перехват которого привел к необходимости выбрасывания исключения `ИсключениеЧтенияФайла`.

Описание пользовательского исключения допускает создание собственных конструкторов. Но для исключения `ИсключениеЧтенияФайла` собственный конструктор не нужен, т.к. единственное, что можно сделать в этом случае - поменять местами имя файла и описание исключения. Переопределять конструктор стоит тогда, когда создаваемое исключение содержит несколько полей со специфическими данными и автоматически создаваемый конструктор по каким-то причинам не устраивает.

Описание исключения

Каждое исключение обладает следующими стандартными свойствами:

Поле	Описание
Описание	Тип: `Строка`. Содержит текстовое представление ошибки, описываемой данным типом исключения.
Причина	Тип: `Исключение?`. Содержит исключение, которое непосредственно послужило причиной возникновения данного исключения.

Поле	Описание
ПодавленныеИсключения	Тип: ФиксированныйМассив объектов типа Исключение. Содержит список подавленных (перехваченных) исключений, который привел к данному.
ПоследовательностьВызовов	Тип: Строка. Текстовое представление стека вызовов, начиная от самой первой ошибки.

Работа с закрываемыми ресурсами

Если в одной области видимости требуется отслеживать состояние нескольких закрываемых объектов, то все эти объекты необходимо явно объявить с использованием модификатора исп. При выходе из области видимости система автоматически закроет все объекты, объявленные с таким модификатором.

Инициализация закрываемых объектов выполняется в порядке их (объектов) объявления. Закрытие объектов выполняется в порядке, обратном порядку инициализации. Если при работе с закрываемым ресурсом возможно возникновение перехватываемого исключения, то правильно будет реализовать саму работу в блоке безопасного кода оператора попытка.

Поведение системы зависит от того, сколько закрываемых ресурсов используется в безопасном коде оператора попытка. Если в безопасном коде используется один закрываемый ресурс, то:

- Если инициализация закрываемого ресурса выполнена с ошибкой Искл1 - весь блок попытка попытка завершается преждевременно по причине выбрасывания исключения Искл1.
- Если инициализация ресурса завершена успешно, а блок безопасного кода завершает преждевременно с исключением Искл1, то:
  - Если закрытие ресурса выполняется успешно, то блок безопасного кода завершается преждевременно по причине выбрасывания исключения Искл1.
  - Если автоматическое закрытие ресурса завершается исключением Искл2, то блок безопасного кода завершается преждевременно по причине выбрасывания исключения Искл1, а исключение Искл2 попадает в список подавленных исключений Искл1.
- Если и инициализация ресурса и его использование завершено успешно, а закрытие ресурса завершается преждевременно с выбрасывания исключения Искл1, то безопасный код завершает "преждевременно" с выбрасыванием исключения Искл1.

Если в безопасном коде используется несколько закрываемых ресурсов, то:

- Если инициализация какого-либо ресурса завершается исключением Искл1, то:
  - Если автоматическое закрытие всех ранее инициализированных ресурсов завершено успешно, то безопасный блок завершается преждевременно с выбрасыванием исключения Искл1.
  - Если автоматическое закрытие всех ранее инициализированных ресурсов завершается преждевременно с выбрасыванием исключений Искл2 - ИсклN, то безопасный блок завершается преждевременно с выбрасыванием исключения Искл1, а исключения Искл2 - ИсклN будут добавлены в список подавленных исключений для исключения Искл1.
- Если инициализация всех ресурсов завершилась успешно, а исключение Искл1 возникло в блоке безопасного кода, то:
  - Если автоматическое закрытие всех ранее инициализированных ресурсов завершено успешно, то безопасный блок завершается преждевременно с выбрасыванием исключения Искл1.
  - Если автоматическое закрытие всех ранее инициализированных ресурсов завершается преждевременно с выбрасыванием исключений Искл2 - ИсклN, то безопасный блок завершается преждевременно с выбрасыванием исключения Искл1, а исключения Искл2 - ИсклN будут добавлены в список подавленных исключений для исключения Искл1.

- Если безопасный блок завершился успешно, то:
  - Если автоматическое закрытие всех, кроме одного, ранее инициализированных ресурсов завершено успешно, то безопасный блок завершается "преждевременно" с выбрасыванием исключения, которое возникло при аварийном закрытии единственного ресурса.
  - Если преждевременно завершается закрытие более одного закрываемого ресурса, то безопасный блок завершается "преждевременно" с выбрасыванием того исключения, которое при закрытии ресурсов возникло первым, а остальные исключения будут размещены в списке подавленных исключений первого исключения. Следует помнить, что закрытие ресурсов (в том числе автоматическое) выполняется в порядке, обратном объявлению ресурсов. Таким образом - "первым" будет исключение, которое возникло при закрытии **последнего** закрываемого ресурса в порядке объявления.

Возможно вложенное использование конструкций обработки исключений.

Пример использования закрываемого объекта

```
метод Скрипт()
    попытка
        пер Файл = Файлы.Создать(СредаИсполнения.ПолучитьПеременную("temp")
 + "\\test.txt")
        исп Поток = Файл.ОткрытьПотокЗаписи()
        Поток.Записать("1-я строка\н")
        Поток.Записать("2-я строка\н")
        Поток.Записать("3-я строка\н")
        Консоль.Записать("Файл записан!")
    поймать Искл: Исключение
        // попадем сюда, если хоть какое-то исключение случилось
        // здесь поток записи уже закрыт
        // переменная Поток здесь уже отсутствует
        Консоль.Записать(Искл.ПоследовательностьВызовов)
        Консоль.ЗаписатьСтроковоеПредставление(Искл)
    вконце
        // попадем сюда в любом случае
        // здесь поток записи уже закрыт!
        // переменная Поток здесь уже отсутствует
        Консоль.Записать("Файловый поток закрыт, переменная недоступна
 (выход из области видимости)")
    ;
;
```

Встроенные типы исключений

Ниже перечислены исключения, которые могут выбрасываться во время работы программы. Если для какого-либо типа исключения указано наличие конструктора, то это означает, что предоставляется возможность самостоятельно выбросить исключение указанного типа и для такого исключения определены следующие конструкторы:

- Без параметров.
- Только описание.
- Только причина исключения (при необходимости создать вложенное исключение).
- Описание и причина исключения.

Таблица **7.** Типы исключений языка

Имя	Конструктор	Описание
ИсключениеАрифметики		Исключение, возникающее при выполнении арифметических операций. Например, данное исключение возникнет при выполнении операции деления на 0.

Имя	Конструктор	Описание
ИсключениеПроверкиТипа		Исключение выбрасывается при несоответствии типа формального параметра типу фактического значения.
ИсключениеСравнения		Исключение возникает при попытке сравнения объектов, которые нельзя сравнивать.
ИсключениеДинамическогоВыполнения		Исключение, которое выбрасывается при выполнении динамического кода.
ИсключениеНедопустимыйАргумент	Да	Исключение выбрасывается в том случае, если аргумент не удовлетворяет ограничениям.
ИсключениеНедопустимыйФормат	Да	Исключение выбрасывается в том случае, если аргумент имеет тип Строка и нарушен формат, который ожидается в этой строке. Например, при формировании форматной строки.
ИсключениеНедопустимоеСостояние	Да	Исключение выбрасывается в том случае, если действие выполняется для объекта, состояние которого не поддерживается данным методом.
ИсключениеИндексВнеГраниц	Да	Исключение выбрасывается если переданное значение индекса выходит за допустимые рамки.

Соглашения при написании кода

В этом разделе перечислены требования и рекомендации, которых мы советуем придерживаться при оформлении программного кода и при использовании некоторых инструкций языка.

Требования при написании кода

В этом разделе перечислены правила, которые мы считаем обязательными при написании кода.

Общее оформление

Синтаксический отступ

Блоки вложенных инструкций, находящиеся на одном уровне, выделяются одним синтаксическим отступом.

Правильно	Неправильно

`метод Скрипт()`
 	
`    пер Переменная1: Строка`
 	
`    для Счетчик = 1 по 5`
`        Метод1()`
`    ;`
 	
`;`
	`метод Скрипт()`
	
	`пер Переменная1: Строка`
	
	`для Счетчик = 1 по 5`
	`Метод1();`
	
	`;`

Один синтаксический отступ составляет 4 пробела. Символы табуляции не используются.

Длина строки

Максимальная длина строки — 120 символов.

Совет: Допускаются строки большей длины, если разбиение таких строк на части по 120 символов ухудшает их чтение и понимание.

Пустые строки

Использование пустых строк не регламентируется.

Правильно

`метод Скрипт()`
`    пер Переменная1: Строка`
`    для Счетчик = 1 по 5`
`        Метод1()`
`    ;`
`;`

`метод Скрипт()`
 	
`    пер Переменная1: Строка`
 	
`    для Счетчик = 1 по 5`
`        Метод1()`
`    ;`
 	
`;`

Составные инструкции

Составные инструкции завершаются символом «;», который пишется на отдельной строке с тем же синтаксическим отступом, что и сама инструкция.

Правильно	Неправильно

`для Счетчик = 1 по 5`
`    Метод1()`
`    Метод2()`
`;`
	`для Счетчик = 1 по 5`
	`    Метод1()`
	`    Метод2();`

`если Переменная1 > 5`
`    Метод1()`
`    Метод2()`
`иначе`
`    Метод3()`
`;`
	`если Переменная1 > 5`
	`    Метод1()`
	`    Метод2()`
	`иначе`
	`    Метод3();`

`метод Метод1(): Булево`
`    пер Переменная1 = Истина`
`    возврат Переменная1`
`;`
	`метод Метод1(): Булево`
	`    пер Переменная1 = Истина`
	`    возврат Переменная1;`

Имена

Все имена, которые вы используете для переменных, типов и методов, пишутся в стиле CamelCase — несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы. Если слово одно — оно тоже пишется с прописной буквы.

Правильно	Неправильно

`перечисление СтепеньВажности`
`    Высокая,`
`    Обычная,`
`    Низкая`
`;`
	`перечисление Степень_Важности`
	`    высокая,`
	`    обычная,`
	`    низкая`
	`;`

`структура ВходящееСообщение` `;`	`структура входящееСообщение` `;`

`метод Версия()`
`    пер МояПеременная: Число`
`    пер РабочийКаталог: Строка`
`;`
	`метод версия()`
	`    пер Моя_переменная: Число`
	`    пер рабочий-каталог: Строка`
	`;`

В аббревиатурах только первая буква заглавная — Xml, Json, Uuid.

Правильно	Неправильно
`пер ДокументXml: Строка` `пер СтрокаJson: Строка`	`пер ДокументXML: Строка` `пер СтрокаJSON: Строка`

Имена констант пишутся БОЛЬШИМИ_БУКВАМИ_С_ПОДЧЕРКИВАНИЯМИ.

Правильно	Неправильно
`конст ВЕРСИЯ_СЕРВЕРА = 1.1` `конст ТАЙМАУТ_СЕРВЕРА = 60с`	`конст ВерсияСервера = 1.1` `конст Таймаут_сервера = 60с`

Типы исключений называются с префиксом «Исключение».

Правильно	Неправильно

`исключение ИсключениеЧтенияФайла`
`    пер ИмяФайла: Строка`
`;`
	`исключение ЧтенияФайла`
	`    пер ИмяФайла: Строка`
	`;`

Кроме этого существует рекомендация по именам перечислений.

Перенос выражений

Выражения переносятся в тех случаях, когда:

- инструкция, содержащая выражение, превышает максимальную длину строки,
- перенос выражения улучшает понимание инструкции.

В каждой перенесенной строке может содержаться одна или несколько операций.

Операции пишутся в начале перенесенных строк.

Правильно	Неправильно

`если Переменная1 > 5`
`     или Переменная2 < Переменная3`
`     и не Переменная4`
`    Метод1()`
`    Метод2()`
`;`
	`если Переменная1 > 5 или`
	`     Переменная2 < Переменная3 и`
	`     не Переменная4`
	`    Метод1()`
	`    Метод2()`
	`;`

`Консоль.Записать(Переменная1`
`                + Переменная2`
`                - Переменная3)`
	`Консоль.Записать(Переменная1 +`
	`                Переменная2 -`
	`                Переменная3)`

Совет: При конкатенации строк допускается писать операцию в конце строки:

Допускается

`Консоль.Записать("Сегодня " +`
`                НомерДняНедели +`
`                "-й день недели")`

Перенесенные строки выравниваются:

- либо по началу первого операнда,

Правильно	Неправильно

`если Переменная1 > 5`
`  или Переменная2 < Переменная3`
`  и не Переменная4`
`Метод1()`
`Метод2()`
`;`
	`если Переменная1 > 5`
	` или Переменная2 < Переменная3`
	`и не Переменная4`
	`Метод1`
	`Метод2()`
	`;`

`Консоль.Записать("Сегодня " +`
`НомерДняНедели +`
`"-й день недели")`
	`Консоль.Записать("Сегодня " +`
	`НомерДняНедели +`
	`"-й день недели")`

- либо одним синтаксическим отступом.

Правильно	Неправильно

`если Переменная1 > 5`
`или Переменная2 < Переменная3`
 	
`Метод1()`
`Метод2()`
`;`
	`если`
	`Переменная1 > 5`
	`или Переменная2 < Переменная3`
	
	`Метод1()`
	`Метод2()`
	`;`

`Консоль.Записать("Сегодня " +`
`НомерДняНедели +`
`"-й день недели"`
`)`
	`Консоль.Записать(`
	`"Сегодня " +`
	`НомерДняНедели +`
	`"-й день недели")`

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 перенесенной строке может содержаться один или несколько параметров.

Запятые, разделяющие параметры, пишутся в концах строк.

Правильно

`Массив.Добавить(Свойство.Параметр1,`
`Свойство.Параметр2,`
`Свойство.Параметр3,`
`Свойство.Параметр4)`

`Массив.Добавить(Свойство.Параметр1, Свойство.Параметр2,` `Свойство.Параметр3, Свойство.Параметр4)`

Перенесенные параметры выравниваются:

- Либо по началу первого параметра. В этом случае закрывающая скобка пишется в конце последней перенесенной строки,

Правильно	**Неправильно**

`Массив.Добавить(Свойство.Параметр1,`
`  Свойство.Параметр2,`
`  Свойство.Параметр3,`
`  Свойство.Параметр4)`
	`Массив.Добавить(Свойство.Параметр1,`
	`  Свойство.Параметр2,`
	`  Свойство.Параметр3,`
	`  Свойство.Параметр4)`

`метод Метод1(Параметр1: Число,`
`  Параметр2: Строка,`
 	
`  Параметр3: Строка): Булево`
 	
`  пер Переменная1 = 40`
`  возврат Истина`
`;`
	`метод Метод1(Параметр1: Число,`
	`  Параметр2: Строка,`
	`  Параметр3: Строка): Булево`
	
	`  пер Переменная1 = 40`
	`  возврат Истина`
	`;`

- Либо одним синтаксическим отступом. В этом случае параметры переносятся начиная с первого, а закрывающая скобка пишется
  - либо в конце последней перенесенной строки,
  - либо на отдельной строке с отступом, соответствующим отступу всей инструкции.

Правильно

`Массив.Добавить(`
`  Свойство.Параметр1,`
`  Свойство.Параметр2,`
`  Свойство.Параметр3,`
`  Свойство.Параметр4`
`)`
 	
`Массив.Добавить(`
`  Свойство.Параметр1,`
`  Свойство.Параметр2,`
`  Свойство.Параметр3,`
`  Свойство.Параметр4)`

`метод Метод1(`
`  Параметр1: Число,`
`  Параметр2: Строка,`
`  Параметр3: Строка`
`): Булево`
 	
`  пер Переменная1 = 40`
`  возврат Истина`
`;`
 	
`метод Метод1(`
`  Параметр1: Число,`
`  Параметр2: Строка,`
`  Параметр3: Строка): Булево`
 	
`  пер Переменная1 = 40`
`  возврат Истина`
`;`

Литералы коллекций переносятся по тем же правилам, что и параметры методов.

Правильно

```
пер ИменаПолей = ["Идентификатор",
                  "Размер",
                  "Цвет"]

пер ИменаПолей = [
    "Идентификатор",
    "Размер",
    "Цвет"
]
```

```
пер МножествоСДанными = {1,
                         2,
                         3}

пер МножествоСДанными = {
    1,
    2,
    3}
```

```
пер КурсыВалют = {"RUB": 1,
                  "BYN": 31.01}

пер КурсыВалют = {
    "RUB": 1,
    "BYN": 31.01
}
```

Операции и инструкции

Объявления методов

Необязательные параметры (параметры со значениями по умолчанию) должны располагаться после обязательных параметров (без значений по умолчанию).

Правильно

```
метод НовоеСообщение(Вид: Строка,
                     Дата =
 ""): Соответствие
    возврат {"тип": Вид,
 "дата": Дата}
;
```

Неправильно

```
метод НовоеСообщение(Дата = "",

  Вид: Строка): Соответствие
    возврат {"тип": Вид,
 "дата": Дата}
;
```

В объявлениях экспортируемых методов ключевое слово `экспорт` пишется на отдельной строке.

Правильно

```
экспорт
метод Тест()
;
```

Неправильно

```
экспорт метод Тест()
;
```

Описание типа и инициализация

Синтаксис

Тип отделяется от объявления двоеточием и пробелом. Перед двоеточием пробел не ставится.

Правильно	Неправильно
`пер Переменная1: Строка`	`пер Переменная1 : Строка`

Составной тип

Рядом с разделителем составных типов пробел не ставится.

Правильно	Неправильно
`пер Переменная1: Строка|Число|Булево = 0`	`пер Переменная1: Строка | Число | Булево = 0`

Тип **Неопределено**

Тип Неопределено в списке типов явно не указывается, вместо него используется сокращение «?», которое пишется:

- слитно с предыдущим типом, если типов всего 2,
- через символ «|», если типов больше двух.

Правильно	Неправильно
`пер Переменная1: Строка?`	`пер Переменная1: Строка|?`
`пер Переменная1: Строка|Число|?`	`пер Переменная1: Строка|Число?`

Инициализация

Не следует явно указывать типы переменных, инициализируемых литералами.

Правильно	Неправильно
`пер Переменная1 = "значение"`	`пер Переменная1: Строка = "значение"`

Не следует явно указывать типы переменных, инициализируемых конструктором

Правильно	Неправильно
`пер Настройки = новый Массив()`	`пер Настройки: Массив = новый Массив()`

Проверка логических значений

Логические значения, например, результат логического выражения, значение, возвращаемое методом, значения переменных типа Булево, не следует проверять путем сравнения их с литералами Истина и Ложь.

Правильно	Неправильно

пер Переменная1 = Истина
 	
если Переменная1
    Переменная2 = 50
;
	пер Переменная1 = Истина
	
	если Переменная1 == Истина
	    Переменная2 = 50
	;

метод Скрипт()
    пер Переменная3: Строка
 	
    если не Метод1()
        Переменная3 = "проверка"
    ;
;
 	
метод Метод1(): Булево
    возврат Ложь
;
	метод Скрипт()
	    пер Переменная3: Строка
	
	    если Метод1() == Ложь
	        Переменная3 = "проверка"
	    ;
	;
	
	метод Метод1(): Булево
	    возврат Ложь
	;

Проверка значения **Неопределено**

Проверку значения на Неопределено следует выполнять через равенство, а не с помощью операции это.

Правильно	Неправильно

если Значение == Неопределено
 	
  Консоль.Записать("Это Неопределено")
;
	если Значение это Неопределено
	
	  Консоль.Записать("Это Неопределено")
	;

если Значение != Неопределено
 	
  Консоль.Записать("Это другое значение")
;
	если Значение это не Неопределено
	
	  Консоль.Записать("Это другое значение")
	;

Массовая конкатенация строк

Под массовой конкатенацией строк понимается 1000 операций конкатенации строк и более. Это количество может быть и меньше при увеличении длин строк: чем строки длиннее, тем операции выполняются дольше.

В таких случаях следует использовать метод `МенеджерСтрок.Соединить()`. Такой код не только быстрее выполняется, но и приводит к снижению потребления оперативной памяти.

Правильно	Неправильно

`пер Тексты = новый Массив()`
`пер Результат: Строка`
`для НомерКолонки = 1 по 1000`
 	
`  Тексты.Добавить("очередной текст")`
`;`
 	
`Результат`
`  = Строки.Соединить(Тексты,`
 	
`  Символы.ВозвратКаретки)`
	`пер Результат: Строка`
	`для НомерКолонки = 1 по 1000`
	`    Результат = Результат +`
	`                "очередной текст" +`
	
	`  Символы.ВозвратКаретки`
	`;`

Особенно важен этот метод для конкатенации в циклах и в универсальных механизмах, потому что они могут применяться на сколь угодно больших объемах данных.

Кроме этого существуют рекомендации по конкатенации строк.

Операция **это**

Вместо отрицания результата проверки следует использовать операцию проверки с отрицанием.

Правильно	Неправильно

`если Значение это не Строка`
 	
`  Консоль.Записать("Это не строка")`
`;`
	`если не (Значение это Строка)`
	
	`  Консоль.Записать("Это не строка")`
	`;`

Рекомендации при написании кода

В этом разделе перечислены советы, которые могут сделать код более читаемым и удобным.

Имена перечислений

В именах перечислений используйте слово «Вид» (Kind), вместо «Тип» (Type).

Рекомендуется	Не рекомендуется

`перечисление ВидКнопки`
`    Круглая,`
`    Квадратная`
`;`
	`перечисление ТипКнопки`
	`    Круглая,`
	`    Квадратная`
	`;`

Литералы коллекций

Использование литералов коллекций предпочтительнее, чем ручное наполнение.

Рекомендуется	Не рекомендуется

`пер ИменаПолей = ["Идентификатор", "Размер", "Цвет"]`	`пер ИменаПолей: Массив`
	`ИменаПолей.Добавить("Идентификатор")`
	`ИменаПолей.Добавить("Размер")`
	`ИменаПолей.Добавить("Цвет")`

`пер МножествоСДанными = {1, 2, 3}`	`пер МножествоСДанными: Множество`
	`МножествоСДанными.Добавить(1)`
	`МножествоСДанными.Добавить(2)`
	`МножествоСДанными.Добавить(3)`

`пер КурсыВалют = {"RUB": 1, "BYN": 31.01}`	`пер КурсыВалют: Соответствие`
	`КурсыВалют.Вставить("RUB", 1)`
	`КурсыВалют.Вставить("BYN", 31.01)`

Конкатенация строк

Используйте неявное преобразование к типу Строка

Конкатенация со строкой предпочтительнее, чем вызов метода `Строка()` с последующей конкатенацией.

Рекомендуется	Не рекомендуется
`пер Счетчик = 1` `Консоль.Записать("Итерация №" + Счетчик)`	`пер Счетчик = 1` `Консоль.Записать("Итерация №" + Строка(Счетчик))`

Используйте интерполяцию

Не забывайте, что помимо конкатенации вы можете использовать интерполяцию строк. Во многих случаях такая конструкция может выглядеть более понятно.

Рекомендуется	Не рекомендуется

`пер Счетчик = 1`
`пер Всего = 15`
`Консоль.Записать("Итерация № %Счетчик из %Всего")`
	`пер Счетчик = 1`
	`пер Всего = 15`
	`Консоль.Записать("Итерация №" + Строка(Счетчик) + " из " + Строка(Всего))`

Глава 3

Стандартная библиотека

Кроме базовых объектов, которые рассматривались ранее, система 1С:Исполнитель включает *стандартную библиотеку*. Стандартная библиотека - это набор объектов, который является составной частью системы 1С:Исполнитель и позволяет выполнять различные действия, в том числе с внешним, по отношению к системе 1С:Исполнитель окружением.

Стандартная библиотека включает в себя большое количество различных объектов. Сюда входят объекты работы со строками и переменными окружения, с файлами и процессами операционной системы, объекты, позволяющие работать с консолью и т.д. Более подробно все эти (и некоторые другие) возможности будут рассмотрены в других разделах данного документа.

Глава
4

Работа с операционной системой

Данный раздел содержит описание средств взаимодействия с операционной системой.

Т.к. система 1С:Исполнитель является языком сценариев, предназначенным для автоматизации выполнения рутинных задач, возникающих в процессе администрирования информационных систем (в том числе, основанных на базе системы "1С:Предприятие"), то кажется очевидным, что такая система должна уметь взаимодействовать с операционной системой, в которой исполняется сценарий.

Данный раздел будет посвящен как раз такому взаимодействию. Будут рассмотрены объекты, позволяющие делать следующее:

- Взаимодействовать с консолью операционной системы (см. Консоль операционной системы).
- Работать с переменными окружения самой операционной системы и виртуальной машиной Java (см. Переменные окружения).
- Использовать файлы, хранящиеся на дисках компьютера (см. Работа с файлами).
- Взаимодействовать с процессами операционной системы (см. Процессы операционной системы).

Понятия, связанные с данным

Что такое 1С:Исполнитель

Аргументы командной строки

Данный раздел описывает, каким образом сценарий может получить данные, необходимые для своей работы, не требуя интерактивного вмешательства пользователя.

При запуске сценария может потребоваться передать этому сценарию некоторые входные данные. Например, файлы в каком каталоге на диске надо обработать. Это можно сделать интерактивно, путем явного запроса значения у пользователя с использованием консоли (см. Консоль операционной системы). Но это далеко не всегда удобно. Поэтому есть еще один вариант такой передачи аргументов - указание их в командной строке запуска сценария (см. Командная строка 1С:Исполнитель). Аргументы командной строки могут называться по-разному: если они всего-лишь уточняют поведение сценария (указывают рабочий каталог и обрабатываемый файл), тот в этом случае мы будет говорить, что "сценарий получает параметры". Если сценарий может выполнять различные действия, каждое действие имеет свое мнемоническое имя, а для каждого (или некоторых) действий нужны какие-либо уточнения, то в этом случае мы будем говорить, что "у сценария есть команды, для которых требуются параметры". Но вполне допустимо и употребление термина "аргумент" как обобщения всего вышесказанного.

Параметры сценария - это параметры метода `Скрипт()` данного сценария. В командной строке сценария параметры должны быть указаны в том порядке, в каком они (параметры) указаны в описании метода `Скрипт()`. При указании параметры необходимо соблюдать типы параметров. Для передачи из командной строки поддерживаются значения типа `Строка`, `Число` и `Булево`. Отсутствие какого-либо параметра приведет к ошибке.

Допустим, что у нас есть сценарий с единственным методом `Скрипт()`, у которого описаны три параметра:

```
метод Скрипт(Параметр1: Число, Параметр2: Строка, Параметр3: Булево)
    Консоль.Записать("Параметр 1: значение =  " + Строка(Параметр1) + ", тип
 = " + Строка(Параметр1.ПолучитьТип()))
    Консоль.Записать("Параметр 2: значение =  " + Строка(Параметр2) + ", тип
 = " + Строка(Параметр2.ПолучитьТип()))
    Консоль.Записать("Параметр 3: значение =  " + Строка(Параметр3) + ", тип
 = " + Строка(Параметр3.ПолучитьТип()))
;
```

Первый параметр имеет тип `Число`, второй - тип `Строка`, а последний параметр имеет тип `Булево`. Теперь при запуске этого сценария, система 1С:Исполнитель будет выполнять проверку количества и типов передаваемых параметров.

Консоль операционной системы

Содержит описание работы с консолью операционной системы с помощью объекта `Консоль`.

В информационных технологиях под термином *консоль* понимается совокупность устройств ввода/вывода, обеспечивающих диалог человека и компьютера. В данном документе будет использовать более упрощенный вариант этого термина. Под термином *консоль* мы будем понимать символьный интерфейс ввода/вывода, который система 1С:Исполнитель "наследует" у командного интерпретатора операционной системы. Для работы с консолью предназначен объект глобального контекста `Консоль`.

Как и любой интерфейс ввода/вывода, консоль предполагает два основных действия: вывести информацию в консоль и получить информацию из консоли. В этот момент сделаем небольшое отступление и кратко упомянем такое понятие, как *стандартные потоки*. С точки зрения программы, стандартные потоки - это предопределенные интерфейсы к механизмам операционной системы, позволяющие выполнять три стандартных действия:

1. Выводить информацию в консоль. Если мы хотим сообщить пользователю какую-то информацию о работе нашей программы - мы будем использовать именно это действие. Используется стандартный поток вывода, также известный как *stdout*.

2. Получать информацию из консоли. Получение информации из консоли будет нужно нам всякий раз, когда надо что-то получить от пользователя: какой-то параметр, ответ на вопрос и т.д. Используется стандартный поток ввода, также известный как *stdin*.
3. Выводить пользователю информацию об ошибках. Когда в исполняемом скрипте возникает ошибка, и эта ошибка доходит до самого интерпретатора системы 1С:Исполнитель, то информация о такой ошибке выводится в стандартный поток вывода информации об ошибках и прочих диагностических сообщениях. Если во время своей работы сценарий обнаруживает какую-то ошибку - информация об этом также должна выводиться в стандартный поток сообщений об ошибка. Этот стандартный поток вывода также известен под именем *stderr*.

Каждый из упомянутых потов может быть переадресован средствами операционной системы. И с этой точки зрения, мы можем хотеть обычный вывод нашего скрипта поместить в один файл, а все ошибки, возникающие во время работы - в другой. С этой точки зрения важно четко разделять, что и куда выводит наш сценарий.

Вернемся к нашему интерфейсу с пользователем.

Для того, чтобы сообщить какую-то информацию пользователю, следует использовать методы, имя которых начинается с префикса `Записать`:

- `Записать()` - выводит в стандартный поток вывода свой параметр.
- `ЗаписатьСтроковоеПредставление()` - выводит в стандартный поток вывода строковое представление объекта, указанное в качестве параметра. Синонимом данному методу служит конструкция вида `Консоль.Записать(Строка(Объект))`.
- `ЗаписатьОшибку()` - данный метод выводит в сообщение об ошибке в стандартный поток сообщений об ошибках.

Для того, чтобы получить от пользователя какую-либо информацию, следует использовать методы, имя которые начинается с префикса `Считать`:

- `СчитатьСимвол()` - позволяет получить от пользователя 1 символ.
- `СчитатьСтроку()` - позволяет получить от пользователя несколько символов (строку).
- `СчитатьЧисло()` - позволяет получить от пользователя числовое значение. Формат ввода определяется локалью, используемой консолью.
- `СчитатьБулево()` - выполняет запрос у пользователя значения типа `Булево`. Фактически, пользователю предлагается ввести какое-либо из значений: `Истина`, `Ложь`, `True`, `False` (ввод является регистрочувствительным).

Работа с консолью

```
метод Скрипт()
    пер Имя: Строка
    пер Возраст: Число
    пер Уверенность: Булево

    пока Имя != "Все"
        Имя = Консоль.СчитатьСтроку("Введите ваше имя: ")
        выбор Имя
        когда "Все"
            прервать
        когда == "Ошибка"
            Консоль.ЗаписатьОшибку("Введено ошибочное имя!")
            продолжить
        иначе
            Возраст = Консоль.СчитатьЧисло("Сколько вам лет: ")
            Уверенность = Консоль.СчитатьБулево("Вы уверены, что вас зовут "
 + Имя + " ")
            если не Уверенность
                продолжить
            ;
            Консоль.Записать("Вас зовут " + Имя + ", и ваш возраст равен "
 + Возраст)
```

```
        ;
    ;
    Консоль.Записать("Работа завершена")
;
```

Работа с файлами

Содержит описание инструментов работы с файлами в системе 1С:Исполнитель.

Общая информация

Все данные на компьютере хранятся в файлах: тестовых, графических, файлами с базами данных и т.д. Практически любое действие, для которого будет писаться сценарий на встроенном языке, будет содержать операции с файлами. В файлы будут записываться журнал выполнения каких-либо действий, в них хранятся настройки и т.д.

Любой файл (или каталог) характеризуется путем к нему. Путь бывает полным или относительным. Полный путь указывает местоположение файла независимо от того, в каком месте файловой системы мы сейчас находимся. Относительный путь указывает расположение файл относительно рабочего каталога пользователя или текущего каталога приложения. В любом случае, путь содержит список каталогов, в порядке вложенности этих каталогов, указывающих расположение файла или каталога. Каталоги в описании пути разделяются специальным символом, который называется *разделитель*. Разные операционные системы используют разные символы в качестве разделителя пути. Однако, система 1С:Исполнитель является кроссплатформенной системой, поэтому один и тот же сценарий должен без ошибок работать на любой поддерживаемой операционной системе. В связи с этим при работе с файлами необходимо помнить о следующих особенностях:

1. Если путь к файлу задается литералом, то допускается использовать любой разделитель пути, который в принципе допустим на поддерживаемых операционных системах. 1С:Исполнитель автоматически выполнит установку корректного разделителя для используемой ОС. В этом смысле пути к файлам являются независимыми от используемой ОС. Однако, фактические пути, которые будут получены, например, при использовании объекта `Файл`, будут содержать разделители пути используемой операционной системы. И в этом смысле пути к файлам являются зависимыми от используемой операционной системы.
2. Если необходимо выполнять анализ пути к файлу или каталогу, то следует использовать тот символ разделителя, который принят в той операционной системе, где используется сценарий. Этот разделитель можно получить с помощью свойства `Файлы.СимволРазделителя`.
3. Необходимо помнить, что понятие полного пути на разных операционных системах может существенно различаться. Поэтому желательно передавать нужные пути или в качестве параметров сценария или указывать их через переменные окружения (см. Переменные окружения).

Файл

Во встроенном языке файл описывается с помощью одноименного объекта `Файл`. С помощью этого объекта с реальным файлом нельзя выполнить практически никаких действий, но зато этот объект предоставляет доступ к информации о файле.

Объект `Файл` позволяет получить следующую информацию о файле:

- Имя файла - свойство `Имя`.
- Имя файла без расширения - свойство `ИмяБезРасширения`.
- Расширение файла - свойство `Расширение`.
- Полное имя файла (включая путь) - свойство `Путь`.
- Каталог, в котором расположен файл - свойство `Каталог`.
- Размер файла - свойство `Размер`.

- Признак скрытого файла - свойство `Скрытый`. Файл считается скрытым, если:
  - ОС Linux: имя файла начинается с символа "`.`" (точка).
  - ОС macOS: имя файла начинается с символа "`.`" (точка).
  - ОС Windows: для файла установлен атрибут `Скрытый` (`Hidden`).
- Время последнего изменения файла - свойство `ВремяИзменения`.
- Список подчиненных файлов (если объект `Файл` описывает каталог) данного - свойство `Дочерние`.

Также объект `Файл` позволяет проверить следующую информацию о файле, связанным с объектом:

- Это файл? С помощью метода `ЯвляетсяФайлом()`.
- Это каталог? С помощью метода `ЯвляетсяКаталогом()`.
- Это ссылка? С помощью метода `ЯвляетсяСсылкой()`. Для ОС Windows под термином "ссылка" подразумевается так называемая symbolic link.

Также объект `Файл` позволяет работать с тремя основными правами работы с файлом: чтения файла, записи в файл и возможность запуска файла. Это делается с помощью следующих методов:

- Проверка возможности выполнения действия: `ЕстьПраво()`.
- Установка возможности выполнения действия: `УстановитьПрава()`.
- Получить права для текущего пользователя операционной системы: `ПолучитьПрава()`.

Последней возможностью, которую предоставляет объект `Файл` является возможность получить для файла объекты для чтения из файла и записи в файл с помощью методов `ОткрытьПотокЧтения()` и `ОткрытьПотокЗаписи()`.

Работа с файлами

Для того, чтобы иметь возможность выполнять с файлами какие-либо действия, предназначено менеджер работы с файлами, который доступен через свойство глобального контекста `Файлы`. Менеджер работы с файлами предоставляет достаточно широкие возможности для работы. Обращение к методам этого менеджера будет выглядеть как `Файлы.ИмяВызываемогоМетода()`. Для упрощения, в данной главе имя свойства будет опускаться, но в примерах, очевидно, будет использоваться "правильный" вызов метода.

В случае необходимости создать файл или каталог, необходимо использовать методы, соответственно, `Создать()` и `СоздатьКаталог()`. Используя данные методы следует учитывать, что эти методы автоматически создадут все каталоги, указаны в имени создаваемого файла (или каталога), но которые отсутствуют в реальной файловой системе.

Менеджер работы с файлами предоставляет набор методов для выполнения базовых функций работы с файлами:

- Копирование файла - `Скопировать()`.
- Перемещение файлов - `Переместить()`.
- Переименовать файл - `Переименовать()`.
- Удалить файл - `Удалить()`.
- Найти файлы - `Найти()`. Поиск файлов настраивается с помощью специального объекта `НастройкаПоискаФайлов`. Этот объект позволяет указать что и как будет искать метод (см. Настройка поиска файлов).
- Проверить, что указанный каталог не содержит файлов - `КаталогПустой()`.
- Получить домашний каталог текущего пользователя - `ПолучитьДомашнийКаталог()`.

Менеджер также предоставляет базовый набор методов по работе с временными файлами и каталогами:

- Создание временного файла - `СоздатьВременныйФайл()`. Для временного файла можно указать префикс и суффикс имени временного файла, что позволяет более четко временные файлы по задаче/приложению. Также предоставляется возможность указать. что временный файл должен быть удален после завершения работы текущего процесса.

- Создание временного каталога - СоздатьВременныйКаталог(). Как и метод создания временного файла, имеется возможность указать для временного каталога особый префикс и необходимость удалить каталог после окончания работы текущего процесса.

И временный файл и временный каталог будут создаваться в каталоге временных файлов, который указан для процесса файловой системы, в котором работает интерпретатор встроенного языка. Каталог временных файлов может быть указан, например, с помощью переменной окружения операционной системы.

Настройка поиска файлов

Данный объект необходимо указать методы Найти() в том случае, если настройки конкретного поиска отличаются от настроек по умолчанию. Объект позволяет выполнить следующие настройки:

- Указать глубину поиска в структуре каталогов (относительно каталога, указанного в качестве первого параметра метода Найти()). Например, для поиска только в каталоге Windows (обычно расположенного по пути C:\Windows), без обхода вложенных каталогов, следует указать глубину поиска равной 1. Данный параметр задается с помощью метода МаксимальнаяГлубина().
- Включать или исключать каталоги из поиска. Данный параметр задается с помощью метода ИсключитьКаталоги().
- Включать или исключать обычные файлы из поиска. Данный параметр задается с помощью метода ИсключитьФайлы().
- Включать или исключать из поиска символические ссылки из поиска. Данный параметр задается с помощью метода ИсключитьСсылки().
- Позволяет задать фрагмент текста, который должен встречаться в именах файлов или каталогов. Текст будет применять без учета регистра, будет проверяться вхождение заданного фрагмента в любой части файла. Не поддерживаются символы подстановки. Данный параметр задается с помощью метода ИмяСодержит().
- Определять интервал времени (как закрытый, так и открытый), в который попадает время создания файла или каталога. Под закрытым понимается указание то или иной границы, под открытым - не указание границы. Данный параметр задается с помощью метода Создано(). Если граница указана, то дата и время указанной границы входят в проверяемый интервал. Если какая-либо граница не задана, это означает отсутствие ограничений с этой "стороны".
- Определять интервал времени, когда файла или каталог был изменен. Работает аналогично интервалу создания. Данный параметр задается с помощью метода Изменено().
- Определить, файлы какого размера должны быть найдены. Интервал размера файлов задается диапазоном (открытым или закрытым). Значения границ попадают в проверяемый интервал. Данный параметр задается с помощью метода Размер().

Пример использования файловых методов

```
метод Скрипт()
    пер НастройкиПоиска: НастройкиПоискаФайлов
    НастройкиПоиска = новый НастройкиПоискаФайлов()
        .ИмяСодержит(".")
        .МаксимальнаяГлубина(1)
        .ИсключитьФайлы(Истина)
        .ИсключитьКаталоги(Ложь)
    пер СписокФайлов = Файлы.Найти("c:\\program files\\1cv8\
\", НастройкиПоиска)
    Консоль.Записать("Найдено версий = " + СписокФайлов.Размер())
    для Файл из СписокФайлов
        если Файл.ЯвляетсяКаталогом()
            Консоль.Записать(Файл.Имя)
        ;
    ;
;
```

Переменные окружения

Содержит описание работы с переменными окружения и системными свойствами текущей JVM с помощью объекта СредаИсполнения.

Операционные системы, под управлением которых может работать система 1С:Исполнитель, используют такое понятие как *переменная окружения*. Переменная окружения - это переменная, которая предоставляется операционной системой и хранит некоторое текстовое значение. Некоторые переменные окружения создает собственно операционная система, некоторое может создать пользователь. У виртуальной машины Java (JVM) существует аналог переменных окружения, которые называются *системными свойствами*. Эти свойства частично формируются самой виртуальной машиной при запуске, частично могут быть указаны пользователем в виде параметров запуска виртуальной машины.

Сценарий во время своей работы может использовать различные переменные окружения. Например для получения доступа к каталогу пользователя при работе под управлением ОС Windows, можно использовать переменную окружения *USERPROFILE*. Использование этой переменной окружения позволит получать гарантированный доступ к каталогу пользователя, вне зависимости от того, какой пользователь запустил данный сценарий. Для получения конкретной переменной окружения можно использовать метод СредаИсполнения.ПолучитьПеременную(). Если нужно получить все доступные переменные окружения, то можно использовать метод СредаИсполнения.ПолучитьВсеПеременные(). Переменные окружения можно задавать специально для конкретного сценария, например, для передачи в сценарий списка постоянно задаваемых параметров (если сценарий поддерживает такое действие).

Кроме переменных окружения, сценарий может использовать системные свойства виртуальной машины Java. Схема и назначение использования системных свойств аналогично переменным окружения. Из системных свойств JVM можно получить, например имя используемой операционной системы (как это принято в JVM) или домашний каталог пользователя. Получение домашнего каталога пользователя (свойство *user.home*) через системные свойства JVM будет более универсальным способом, т.к. JVM берет на себя все подробности взаимодействия с используемой операционной системой. Для получения конкретного системного свойства следует использовать метод СредаИсполнения.ПолучитьСвойство(), а для получения всех системных свойств, соответственно, СредаИсполнения.ПолучитьВсеСвойства().

Переменные окружения и системные свойства

```
метод Скрипт()
    // вывод всех переменных окружения
    для Переменная из СредаИсполнения.ПолучитьВсеПеременные()
        Консоль.Записать("Имя: " + Переменная.Ключ + " = "
 + Переменная.Значение)
    ;
    // вывод всех системных свойств
    для Переменная из СредаИсполнения.ПолучитьВсеСвойства()
        Консоль.Записать("Имя: " + Переменная.Ключ + " = "
 + Переменная.Значение)
    ;
    // определение используемой операицонной системы
    пер ИмяОС = СредаИсполнения.ПолучитьСвойство("os.name")
    выбор
    когда ИмяОС.НачинаетсяС("windows", Истина)
        Консоль.Записать("Это Windows")
    когда ИмяОС.Содержит("mac", Истина)
        Консоль.Записать("Это OS X")
    когда ИмяОС.Содержит("nux", Истина)
        Консоль.Записать("Это Linux")
    иначе
        Консоль.Записать("Это неизвестная система")
    ;
    Консоль.Записать("Версия ОС - "
 + СредаИсполнения.ПолучитьСвойство("os.version"))
```

```
;
```

Процессы операционной системы

В этом разделе описаны возможности системы 1С:Исполнитель по работе с процессами операционной системы.

При обсуждении выполнения каких-либо программ под управлением операционной системы, используется термин *процесс*. Процессом будем называть отражение исполняемого файла на ресурсы компьютера: занятая оперативная память, загруженный в эту память исполняемый код и т.д. Система 1С:Исполнитель позволяет работать с процессами операционной системы, но не с любым процессом, а только с теми, которые запущены из текущего сценария. Другими словами, вы не можете подключиться к процессу уже запущенного приложения. В тоже время, если вы сами запускаете какое-либо приложение, то процесс такого приложения вам доступен и вы можете взаимодействовать с этим процессом. Для работа с процессами операционной системы предназначен объект `ПроцессОс`. Важно понимать, что объект `ПроцессОс` не является интерпретатором команд операционной системы.

Работа с процессом состоит из нескольких простых шагов:

1. Создание нового процесса. Новый объект `ПроцессОс` создается с помощью конструктора объекта. Если приложение, с которым будет связан процесс, должно получить на вход какие-то параметры - эти параметры необходимо указать именно при создании объекта `ПроцессОс`. Создание процесса не означает, что приложение будет запущено. Также имеется возможность указать, что стандартные потоки вывода и ошибок можно объединить.
2. Запуск созданного процесса. Для запуска процесса предназначен метод `Запустить()`. Запускаемый процесс наследует переменные окружения родительского процесса. При запуске можно указать текущий каталог для запускаемого процесса с помощью параметра метода `Каталог`.
3. Работа с процессом. Система предоставляет следующие возможности по работе с процессом:
   - Выполнить проверку, что с объектом `ПроцессОс` еще ассоциирован какой процесс операционной системы - `Живой()`.
   - Остановить запущенный процесс с помощью метода `Остановить()`.
   - Получить идентификатор процесса (в терминах используемой операционной системы) с помощью метода `ПолучитьPid()`. Реальный идентификатор процесса будет получен только в том случае, если 1С:Исполнитель работает с использованием Java версии 9 и последующих.
   - Ожидать завершения работы процесса с помощью метода `ОжидатьЗавершения()`.
   - Получение стандартного вывода запущенного приложения. Для этого необходимо получить специальный объект типа `ПотокВывода` от объекта `ПроцессОс` и затем читать из этого потока информацию, которая туда помещается процессом. В помощью этого метода предоставляется возможно реализации сценария, который будет анализировать вывод другого приложения и в режиме "реального" времени принимать на основании анализа какие-то решения. Свойство `ПотокОшибок` даст доступ к стандартному потоку ошибок запущенного процесса.
4. Обработка завершения работы процесса. Для этого предоставляется возможность проанализировать код возврата приложения. Получение кода возврата можно выполнить с помощью метода `ПолучитьКодВозврата()`.

Работа с процессом

```
метод Скрипт()
    пер Процесс = новый ПроцессОс("cmd.exe", ["/c", "dir c:\\windows /
N /4"])
    пер Строка: Строка
    Процесс.Запустить()
    пер Вывод = Процесс.ПотокВывода
    пока Истина
        Строка = Вывод.ПрочитатьКакТекст()
```

```
        если Строка.Пусто()
            прервать
        ;
        Консоль.Записать(Строка)
    ;
;
```

Глава
5

Протоколы передачи данных

- SMTP
- HTTP

Данный раздел содержит описание механизмов использования высокоуровневых протоколов передачи данных.

SMTP

В этом разделе описываются объекты, предназначенные для отправки сообщений электронной почты с использованием протокола SMTP.

Электронная почта это механизм отправки и получения электронных сообщений между пользователями компьютерной сети. Электронное сообщение, с которым работает электронная почта, имеет несколько синонимов, каждый из которых может употребляться с равной вероятностью: *письмо*, *электронное письмо*, *сообщение электронной почты*. Как и в реальной жизни, каждое письмо может быть отправлено одному или нескольким адресатам. В качестве однозначного идентификатора адресата выступает *адрес электронной почты*, который, обычно, имеет следующий вид *имя_пользователя@имя_домена*. Подробное описание формата адреса электронной почты приведено в стандарте RFC 5322.

Для того, чтобы отправить почту, предназначено свойство стандартного пространства имен `КлиентSMTP`. С помощью данного объекта предоставляется возможность отправлять электронную почту используя протокол SMTP. При подключении к серверу электронной почты, поддерживаются следующие виды аутентификации:

- Анонимная аутентификация. Описывается стандартом RFC 4505.
- Простой уровень аутентификации (SASL). Описывается стандартом RFC 4422.
- Аутентификация вида запрос-ответ (CRAM-MD5). Описывается стандартом RFC 2195.

Схема работы выглядит следующим образом:

- Создается сообщение электронной почты. Для этого используется объект `ИсходящееСообщениеЭлектроннойПочты`.
- Для сообщения электронной почты задаются различные получатели сообщения (получатели, получатели копии, скрытые получатели, отправитель и т.д.). Для этого используется объект `АдресЭлектроннойПочты` и соответствующие инструменты объекта для работы с сообщением электронной почты.
- Устанавливаются прочие параметры создаваемого почтового сообщения: тема, содержимое и прочие параметры.
- Перед отправкой электронной почты необходимо указать параметры подключения к серверу электронной почты с помощью одноименного объекта (объект `ПараметрыПодключенияSmtp`).
- С помощью метода `КлиентSMTP.Отправить()` выполняется отправка одного или нескольких сообщений электронной почты. Состояние отправки сообщений можно получить с помощью массива объектов `РезультатОтправкиСообщенияЭлектроннойПочты`, который будет выступать в качестве возвращаемого значение метода отправки.

Объект `АдресЭлектроннойПочты` предназначен для указания не только собственно адреса электронной почты, но и представления этого адреса.

HTTP

Описываются механизмы работы с HTTP.

Общая информация

HTTP - это протокол передачи данных прикладного уровня (описан в стандарте RFC 2616). Основой HTTP является взаимодействие "клиент-сервер". В этом взаимодействии выделяются две стороны:

- Клиент - некоторая система, которая инициирует соединение и посылает запрос.
- Сервер - программное обеспечение, которое ожидает установку соединения от клиента, получает запрос, выполняет некоторые действия и возвращает ответ клиенту.

В основе HTTP-взаимодействия лежит понятие URI (или, в более простом варианте - URL) - универсального идентификатора ресурса, который описан к стандарте RFC 3986. Взаимодействие осуществляется с помощью *запросов*. Каждый запрос содержит несколько частей: *стартовая строка* (определяет тип сообщения),

заголовки (определяет параметры запроса) и *тело сообщения* (дополнительные данные запроса, тело сообщения может быть пустым).

Стартовая строка имеет следующий вид: `Метод URI HTTP/Версия`. Рассмотрим, из чего состоит стартовая строка:

- `Метод` - это определение операции, которую необходимо выполнить с ресурсом, которую описан с помощью URI. Состоит из любых символов US-ASCII (кроме управляющих символов и разделителей). В принципе, название метода может быть любым, но рекомендуется придерживаться набора методов, который описан в спецификации HTTP 1.1 (RFC 2616).
- `URI` - описывает путь до конкретного ресурса на том сервере, который будет выполнять запрос. Запрос будет выполняться над данными, которые адресуются указанным URI. Если метод, указанный в запросе, не требует указания какого-либо ресурса, вместо URI следует указать символ "*".
- `Версия` - указывается версия протокола HTTP. В настоящий момент это версия 1.1.

В ответ на запрос клиента, сервер возвращает ответ, который по структуре аналогичен запросу. Ответ сервера содержит стартовую строку ответа, заголовки и тело сообщения. Тело сообщения является необязательным для ответа. Стартовая строка ответа имеет следующий вид: `HTTP/Версия КодСостояния Пояснение`. Рассмотрим подробнее содержимое стартовой строки ответа:

- `Версия` - номер версии HTTP-протокола, аналогично стартовой строки запроса. В настоящий момент это версия 1.1.
- `КодСостояния` - описывает результат выполнения запроса. Код состояния может сообщать о том, что запрос выполнен успешно, что во время запроса произошла какая-то ошибка или что запрос нужно повторить, изменив параметры запроса.
- `Пояснение` - текстовое пояснение результата выполнения запроса. Данный текст не содержит символов перевода строки и возврата каретки.

Заголовки запроса - это набор именованных параметров запроса. Каждый запрос может содержать различный набор таких параметров. Заголовки делятся на 4 большие части: *общие заголовки* (используются как в запросах, так и в ответах), *заголовки запроса* (используются только в запросах), *заголовки ответа* (используются только в ответах) и *заголовки сущностей* (сопровождают каждую сущность сообщений, используются как в запросах, так и в ответах). Имена заголовков не чувствительны к регистру символов. Запрос может содержать несколько заголовков с одинаковым именем. Такое допускается только в том случае, если несколько значений заголовка могут быть перечислены через символ ",", а также если порядок следования значений в заголовке не изменяет семантику заголовка.

Таким образом HTTP-взаимодействие выполняется следующим образом:

- Клиент устанавливает соединение с сервером.
- Клиент готовит запрос и отправляет этот запрос серверу.
- Сервер принимает запрос и пытается выполнить действия, указанные в запросе.
- Результат выполнения запроса сервер оформляет в виде ответа, где код состояния содержит суть ответа. Затем ответ отправляется обратно клиенту
- Клиент получает ответ и анализирует код состояния. По результату анализа клиент принимает решение о дальнейших действиях.

Система 1С:Исполнитель может выступать только в качестве клиента для протокола HTTP.

Методы **HTTP**-запроса

Стандарт HTTP 1.1 (RFC 2616) описывает набор методов, которыми рекомендуется пользоваться для сохранения соответствия стандарту и пониманию другими разработчиками.

Таблица **8.** Методы **HTTP**-запроса

Метод	**Описание**
`CONNECT`	Устанавливает туннель к серверу, определенному URI из запроса.

Метод	Описание
DELETE	Удаляет данные, которые идентифицируются URI из запроса.
GET	Позволяет запросить содержимое какого-либо ресурса или выполнить какое-либо действие.
HEAD	Позволяет получить метаданные, проверить наличие какого-либо ресурса или узнать, что изменилось с момента последнего обращения. Ответ сервера на содержит тела.
OPTIONS	Используется для определения возможностей веб-сервера или параметров соединения для конкретного ресурса.
PATCH	Аналогичен запросу PUT, но применяется к фрагменту данных ресурса.
POST	Позволяет клиенту передать на сервер какие-то данные. Эти данные, скорее всего, будут обработаны сервером.
PUT	Позволяет клиенту передать на сервер какие-то данные. Эта данные, скорее всего, заменят те данные, которые сейчас адресуются URI из запроса.
TRACE	Возвращает полученный запрос обратно клиенту. Позволяет клиенту узнать, что добавляют или изменяют в оригинальном запросе промежуточные узлы (через которые проходит запрос).

Описание методов, приведенное в таблице, является рекомендуемым поведением. Фактическое поведение целиком и полностью определяется фактической реализацией сервера HTTP-запроса.

Коды состояния

Коды состояния группируются в 5 основных классов (RFC 2616):

Таблица **9.** Классы кодов состояния

Код	Класс	Описание
1xx	Информационный	Информирование о процессе передачи запроса.
2xx	Успех	Информирование о том, что запрос успешно выполнился.
3xx	Перенаправление	Сообщает клиенту о том, что для выполнения запроса необходимо выполнить запрос с другими параметрами. Заголовки ответа содержат информацию о том, что надо изменить в запросе.
4xx	Ошибка клиента	Указывает, что запрос клиента содержит ошибку.
5xx	Ошибка сервера	Указывает, что во время выполнения запроса на сервере произошла ошибка.

Объектная модель

Теперь, когда мы знаем, как устроено клиент-серверное взаимодействие с использованием HTTP-запросов, можно наложить на эту схему объектную модель, которую предоставляет система 1С:Исполнитель.

В качестве основы всей модели выступает свойство стандартного пространства имен `КлиентHttp`. Этот объект содержит базовые параметры HTTP-взаимодействия. В тоже время, нам может потребоваться в один момент времени работать с несколькими серверами. Поэтому объект `КлиентHttp` может создавать новые объекты `КлиентHttp`, которые будут получать все параметры из своего базового объекта, кроме некоторых параметров, которые можно изменить при создании копии. Другими словами - будет создавать измененная копия родительского объекта. Говоря про такую возможность (создание нового объекта на базе существующего) мы будет говорить, что существующий объект выступает в роли *фабрики*. Объект `КлиентHttp`, который предоставляет сама система 1С:Исполнитель, будем называть *базовым* объектом.

Одним из ключевых свойств объекта `КлиентHttp` выступает свойство `БазовыйURL`. Это свойство указывает адрес сервера, к которому будут выполняться запросы нашего клиентского приложения. Но в базовом объекте данное свойство пустое. Поэтому для создания клиента, который сможет обратиться к какому-либо серверу, следует воспользоваться фабрикой клиентов, а именно, методом `КлиентHttp.СБазовымURL()`, где параметром функции будет выступать адрес сервера. Таким образом, создание объекта для работы с конкретным сервером будет выглядеть следующим образом: `пер МойКлиент = КлиентHttp.СБазовымURL("http://example.com")`.

Теперь мы имеем объект, который выступает в роли клиента и нам необходимо выполнить запрос к серверу, который мы указали при создании клиента. Для этого мы используем еще одну фабрику объекта `КлиентHttpа`, а именно - фабрику объектов типа `ЗапросHttp`. Объект `ЗапросHttp` служит для формирования запроса от клиента серверу и получения ответа от сервера. Ответ от сервера поступает в виде объекта `ОтветHttp`. Если мы хотим выполнить запрос, описанный в стандарте HTTP 1.1, то можно использовать функцию вида `СоздатьМетод()`, где `Метод` - это имя HTTP-метода, перечисленного в таблице Таблица 8. Методы HTTP-запроса. В этом случае в функцию необходимо передать только URI на сервере, к которому будет применяться запрос. Если мы хотим указать метод в качестве параметра или использовать нестандартный метод, следует использовать функцию `СоздатьЗапрос()`. В этом случае следует передать и имя метода и URI. Таким образом, для создания запроса к серверу, например, GET-запрос, можно использовать одну из следующих записей:

- `пер Запрос = МойКлиент.ЗапросGET("/path/resource?param1=value1&param2=value")`
- `пер Запрос = МойКлиент.СоздатьЗапрос("GET", "/path/resource?param1=value1&param2=value")`

Получив объект, описывающий запрос, мы можем указать этому запросу нужные заголовки (задав их явно или переопределив уже существующие). Для этого у объекта `ЗапросHttp` есть свойство `Заголовки`, которое предоставляет доступ ко всей коллекции заголовков данного запроса. Свойство имеет тип `ЗаголовкиHttp`.

Также объект `ЗапросHttp` позволяет работать как с одним заголовком (методы `ДобавитьЗаголовок()`/`УстановитьЗаголовок()`/`УдалитьЗаголовок()`), так и с несколькими заголовками сразу (методы `ДобавитьЗаголовки()`/`УстановитьЗаголовки()`/`ОчиститьЗаголовки()`). С помощью методов добавления/установки заголовков нельзя изменять заголовки `Content-Length` и `Transfer-Encoding`. Для изменения заголовка `Content-Length` предназначен метод `УстановитьТипСодержимого()`. Для установки информации о клиентском приложении (заголовок `User-Agent`) предназначен метод `УстановитьUserAgent()`.

Для быстрой установки тела запроса из строки предназначен методы `УстановитьТело()` соответственно. Этот метод нужен в том случае, когда тело запроса содержит много информации и формируется предварительно, в текстовой строке.

После того, как наш запрос готов, его необходимо выполнить. Для этого служит метод функция `ЗапросHttp.Выполнить()`. Вызов этого метода встроенного языка приведет к фактическому выполнению запроса. "Внутри" этого вызова будут выполнены все действия HTTP-взаимодействия. Результат вызова будет помещен в объект `ОтветHttp` и этот объект вернется для анализа и получения данных, которые вернул сервер.

Глава
6

Управление кластером серверов системы "1С:Предприятие"

Содержит описание механизма доступа к кластеру серверов системы "1С:Предприятие" для администрирования этого кластера.

Кластер серверов - это группа из нескольких связанных между собой компьютеров, которые используются как единый компьютерный ресурс. Объединение компьютеров в кластер выполняется на программном уровне. С точки зрения администрирования, кластера серверов системы "1С:Предприятие" состоит из следующих компонентов:

- *Центральный сервер* кластера. Это компьютер, на котором работает главный менеджер кластера. Через центральный сервер кластер выполняются все действия по администрированию кластера. На одном физическом компьютере могут быть запущены несколько центральных серверов разных кластеров. Разные кластеры должны различаться номером IP-порта, по которому выполняется взаимодействие к кластером.
- В кластер входит один или несколько *рабочих серверов*. Каждый рабочий сервер выполняется на выделенном компьютере (и идентифицируется именем этого компьютера).
- На рабочем сервер функционирует один или несколько *рабочих процессов*. Обслуживанием клиентских соединений занимается именно рабочий процесс. Технически, количество рабочих процессов, работающих на одном рабочем сервере, не ограничено. Количество рабочих процессов ограничивают настройки кластера серверов и вычислительные возможности рабочего сервера.
- Также, в состав кластера серверов входит *менеджер кластера*. Менеджер кластера обеспечивает функционирование рабочего сервера и взаимодействие с другими рабочими серверами, входящими в состав кластера. Кроме того, менеджеры кластера обеспечивают функционирование *сервисов кластера*.
- *Сервис кластера* - это программный модуль, выполняющий некоторую ограниченную функциональность. Например, отдельный сервис кластера предназначен для получения доступа к механизму полнотекстового поиска информации.

Совет: Описание устройства кластера серверов можно получить в документации к системе "1С:Предприятие", по этой ссылке.

Механизм доступа к кластеру серверов, который предоставляет система 1С:Исполнитель, позволяет получить доступ ко всем компонентам кластера серверов системы "1С:Предприятие" и выполнить с этими компонентами необходимый набор действий.

Собственно работа с кластером серверов выполняется с помощью сервера администрирования кластера серверов. В состав сервера администрирования входит собственно сервер (исполняемый файл имеет имя `ras`) и утилита командной строки, предназначенной для доступа к серверу (исполняемый файл имеет имя `rac`). Более подробно про сервер администрирования написано в документации к системе "1С:Предприятие" здесь. Система 1С:Исполнитель использует для доступа к серверу администрирования утилиту `rac`, из чего вытекают следующие требования:

- На компьютере, выполняющем роль центрального сервера, должен быть развернут сервер администрирования.
- Сервер администрирования должен быть подключен к кластеру серверов.
- На компьютере, который исполняет сценарий системы 1С:Исполнитель, должна быть установлена утилита командной строки `rac`, с помощью которой выполняется доступ к сервер администрирования.

Сервера кластера

Данный раздел описывает работу с центральным сервером.

Общая информация

Для того, чтобы начать работу с кластером серверов, необходимо подключиться к серверу администрирования центрального сервера кластера. Для этого следует использовать конструктор объекта `АдминистрированиеСервера`. Для подключения необходимо знать адрес сервера администрирования требуемого кластера серверов (в символьной или точечной нотации) и IP-порт, который указан при запуске сервера администрирования системы "1С:Предприятие" (подробнее о запуске сервера администрирования написано здесь). Если для кластера серверов создан администратор сервера (читать здесь), то после подключения к центральному сервер требуется выполнить аутентификацию администратора сервера с помощью метода `АдминистрированиеСервера.ВыполнитьАутентификацию()`. После выполнения аутентификации можно выполнять действия по администрированию кластера серверов. Для запуска сервера администрирования можно использовать сценарий, приведенный далее.

Система 1С:Исполнитель предоставляет следующие возможности:

- Получить список администраторов кластера серверов с помощью метода `АдминистрированиеСервера.ПолучитьАдминистраторов()`. Также предоставляется возможность создать нового администратора с помощью функции `АдминистрированиеСервера.СоздатьАдминистратора()`.
- Получить список кластеров, которые зарегистрированы в данном центральном сервере. Это можно осуществить функцией `АдминистрированиеСервера.ПолучитьКластеры()`. Создание нового кластера возможно с помощью метода `АдминистрированиеСервера.СоздатьКластер()`. Если известен идентификатор кластера серверов, то с помощью функции `АдминистрированиеСервера.ПолучитьКластер()` можно получить объект, предназначенный для управления конкретным кластером серверов.

Запуск сервера администрирования

```
конст ПутиV8 = {"eq": "c:\\Program Files\\1Cv8\\", "ne": "c:\\Program Files
 (x86)\\1Cv8\\"}

метод Скрипт(Версия: Строка, Сервер: Строка, Порт: Число, Архитектура: Число)
    пер АрхитектураОС: Число = ТекущаяАрхитектура()
    пер АрхитектураV8: Число
    пер ПутьV8: Строка

    если Версия.ЧислоВхождений(".") != 3
        Консоль.ЗаписатьОшибку("Номер версии указан некорректно")
        возврат
    ;
    выбор
    когда Архитектура == АрхитектураОС
        АрхитектураV8 = АрхитектураОС
        ПутьV8 = ПутиV8["eq"]
    когда АрхитектураОС == 64
        АрхитектураV8 = Архитектура
        ПутьV8 = ПутиV8["ne"]
    иначе

  выбросить новый ИсключениеНедопустимоеСостояние("64-разрядная платформа
 V8 не может быть установлена на 32-разрядной ОС")
    ;
    пер Образ = новый Файл("%ПутьV8\\%Версия\\bin\\ras.exe")
    если не Образ.Существует()
```

```
  Консоль.ЗаписатьОшибку("Сервер администрирования кластера не обнаружен в каталоге:
 " + Образ.Каталог)
        возврат
    ;
    пер Процесс = новый ПроцессОс(Образ.Путь, ["cluster", "--port="
 + Порт, Сервер])
    Процесс.Запустить(СредаИсполнения.ПолучитьПеременную("temp"))
    если не Процесс.Живой()
        Консоль.ЗаписатьОшибку("Что-то пошло не так. Процесс RAS - не живой
 :(")
    иначе
        Консоль.Записать("Запуск RAS выполнен успешно")
    ;
;

метод ТекущаяАрхитектура(): Число
    пер Архитектура
 = СредаИсполнения.ПолучитьПеременную("PROCESSOR_ARCHITECTURE")
    пер Результат = 0
    выбор Архитектура.ВВерхнийРегистр()
    когда == "AMD64"
        Результат = 64
    когда == "X86"
        Результат = 32
    иначе
        Результат = 32
    ;
    возврат Результат
;
```

Пример получения списка кластеров

```
метод Скрипт()
    пер Сервер = новый АдминистрированиеСервера("localhost", 1545)
    пер СписокКластеров = Сервер.ПолучитьКластеры()
    для текКластер из СписокКластеров
        Консоль.Записать("Кластер " + текКластер.Имя)
    ;
;
```

Рабочие сервера

Как можно узнать в документации, кластер серверов состоит из одно или нескольких *рабочих серверов*. Система 1С:Исполнитель предоставляет возможность получить доступ к текущему составу рабочих серверов кластера, а также создать новый рабочий сервер. Для описания рабочего сервера предназначен объект `АдминистрированиеРабочийСервер`. Описание параметров рабочего сервера приведено в документации к системе "1С:Предприятие".

Для того, чтобы получить доступ к рабочему серверу, можно воспользоваться следующими способами:

1. Получить список всех текущих рабочих серверов. Функция `АдминистрированиеКластера.ПолучитьРабочиеСерверы()` вернет массив объектов `АдминистрированиеРабочийСервер`. Из полученного массива можно выбрать описание требуемого рабочего сервера.
2. Получить описание конкретного рабочего сервера с помощью функции `АдминистрированиеКластера.ПолучитьРабочийСервер()`. В этом случае нужно предварительно знать идентификатор рабочего сервера (типа `Ууид`).
3. Создать новый рабочий сервер с помощью метода `АдминистрированиеКластера.СоздатьРабочийСервер()`.

Важное замечание: Для систем, находящихся в промышленной эксплуатации, не рекомендуется размещать несколько рабочих серверов на одном физическом компьютере.

Если в свойства рабочего сервера вносились изменения (или был создан новый рабочий сервер), то для того, чтобы изменения вступили в силу, нужно выполнить метод `АдминистрированиеРабочийСервер.Записать()`.

Свойство рабочего сервера `АдминистрированиеРабочийСервер.ИдентификаторРабочегоСервера` позволит в дальнейшем получать описание конкретного рабочего сервера напрямую, с помощью метода `АдминистрированиеКластера.ПолучитьРабочийСервер()`. Если на одном физическом компьютере создается несколько рабочих серверов, то у каждого рабочего сервера должен быть уникальный IP-порт (свойство `АдминистрированиеРабочийСервер.Порт`) и диапазон IP-портов (свойство `АдминистрированиеРабочийСервер.ДиапазоныПортов`). Уникальность должна поддерживаться в рамках одного компьютера. Для того, чтобы указать один или несколько диапазонов портов, предназначен метод `АдминистрированиеРабочийСервер.ДобавитьДиапазонПортов()`.

Получение рабочих серверов

```
метод Скрипт()
    пер мойКластер = ПолучитьКластер()
    мойКластер.ВыполнитьАутентификацию("", "")
    пер РабочиеСерверы = мойКластер.ПолучитьРабочиеСерверы()
    для Сервер из РабочиеСерверы
        Консоль.Записать("Рабочий сервер - " + Сервер.Имя)
        Консоль.Записать("\тцентральный сервер: "
 + Сервер.ЦентральныйСервер)
        Консоль.Записать("\ткомпьютер: " + Сервер.Компьютер)
        Консоль.Записать("\тидентификатор: "
 + Сервер.ИдентификаторРабочегоСервера)
    ;
;

метод ПолучитьКластер(): АдминистрированиеКластер
    пер Сервер = новый АдминистрированиеСервера("localhost", 1545)
    возврат Сервер.ПолучитьКластеры()[0]
;
```

Кластер серверов

Данный раздел описывает работу с выбранным кластером серверов.

Совет: Описание свойств кластера серверов можно получить в документации к системе "1С:Предприятие", по этой ссылке.

Для управления конкретным кластером серверов служит объект `АдминистрированиеКластер`, который можно получить с помощью объекта `АдминистрированиеСервера`. Кластер серверов можно получить или сразу, имея его идентификатор, или перебрав коллекцию доступных кластеров.

Чтобы начать управление кластером, необходимо выполнить аутентификацию на этом кластере от имени администратора кластера, если таковой задан. Для этого предназначен метод `АдминистрированиеКластер.ВыполнитьАутентификацию()`.

Свойства кластера доступны через одноименные свойства объекта `АдминистрированиеКластер`. Режим балансировки нагрузки в кластере описывается объектом `АдминистрированиеПриоритетВыбораПроцесса`. Уровень безопасности соединений описывается объектом `АдминистрированиеУровеньБезопасностиСоединений`.

После того, как были изменены какие-либо свойства существующего или вновь созданного кластера, эти изменения необходимо записать. Запись выполняется с помощью функции

АдминистрированиеКластер.Записать(). В качестве результата работы функции будет получен идентификатор кластера (объект типа Ууид).

Объект АдминистрированиеКластер позволяет работать со следующими объектами инфраструктуры кластера серверов системы "1С:Предприятие":

- Администраторы кластера серверов.
- Информационные базы. Подробнее здесь.
- Рабочие процессы кластера. Подробнее здесь.
- Профили безопасности. Подробнее здесь.
- Сеансы и соединения к кластером серверов. Подробнее здесь .
- Менеджеры кластера.
- Блокировки.
- Счетчики потребления ресурсов.

Администраторы кластера

Для кластер серверов можно указать администраторов. Таким образом, чтобы выполнять административные действия над кластером - будет необходимо указать имя пользователя и пароль администратора кластера. Не указав эти параметры будет невозможно администрировать кластер.

Для работы со списком администраторов предоставляется следующий набор методов:

- Выполнить аутентификацию администратора кластера. Используется метод АдминистрированиеКластер.ВыполнитьАутентификацию().
- Получить список администраторов кластера серверов. Используется метод АдминистрированиеКластер.ПолучитьАдминистраторовКластера().
- Создать нового администратора кластера с помощью функции АдминистрированиеКластер.СоздатьАдминистратораКластера(). В этом случае создается объект АдминистрированиеАдминистратор.

Объект АдминистрированиеАдминистратор позволяет задать параметры администратора кластера:

- Имя администратор. Это значение будет необходимо указывать при выполнении аутентификации. Не может содержать пробелы и специальные символы.
- Описание предназначено для того, чтобы дать краткое описание данному администратору кластера. Например, здесь можно указать имя и фамилию администратора, а также его контактные данные.
- АутентификацияСтандартная - этот флажок указывает, что для данного администратора поддерживается стандартная аутентификация. Стандартная аутентификация - это аутентификация с помощью указания имени пользователя и пароля. Имя администратора указано в свойстве Имя, а для указания пароля предназначено свойство Пароль. Таким образом, свойство Пароль нужно указывать только в том случае, если для администратора установлен флажок АутентификацияСтандартная.
- АутентифкацияОС - этот флажок указывает, что для данного администратора поддерживается аутентификация средствами операционной системы. В этом случае следует заполнить параметр ПользовательОС, в которое указывается имя пользователя операционной системы. Если текущий пользователь (который уже аутентифицирован операционной системой) подключается к кластеру серверов - ему не будет требоваться указывать свой пароль, операционная система сообщит кластеру серверов, что это пользователь уже аутентифицирован. При работе под управлением ОС Windows, имя пользователя операционной системы указывается в формате \\<имя домена>\<имя пользователя>.

Окончательно создание администратора (или изменение параметров администратора кластера) происходит после применения функции АдминистрированиеАдминистратор.Записать(). В качестве возвращаемого значения будет получено значение типа Ууид. Это значение является идентификатором созданного администратора.

Информационные базы

В разделе приводится общая информация по работе с информационными базами кластера серверов системы "1С:Предприятие".

В системе "1С:Предприятие" различается понятие *информационная база* и *база данных*. Под термином *информационная база* понимается совокупность следующих сущностей: конфигурация, данные, настройки, различные административные данные, а также база данных, в которой это все хранится. Под термином *база данных* понимается упорядоченный набор структурированной информации (данных), которые хранятся в электронном виде в компьютерной системе. Как правило, с базой данных взаимодействует специальный программный комплекс, который называется *система управления базами данных* или *СУБД*. Таким образом, если упоминается информационная база - речь идет о логическом понятии, связанном с системой "1С:Предприятие". Если упоминается база данных, значит речь идет о хранилище данных. Хранилище данных, скорее всего, будет представлено в виде файла (одного или нескольких), размещенного в файловой системе и к которому имеет доступ только процесс СУБД.

В кластере серверов могут быть создано несколько информационных баз. Для идентификации, каждая информационная база обладает собственным именем, которое должно быть уникально в рамках кластера серверов. Кроме имени, информационная база имеет еще и внутренний идентификатор, который является значением типа `Ууид`. Уникальный идентификатор информационной базы доступен только при программной работы с кластером серверов.

Каждая информационная база описывается специальным объектом системы 1С:Исполнитель `АдминистрированиеИнформационнаяБаза`. Для того, чтобы получить описание конкретной информационной базы, можно воспользоваться следующими способами:

1. Получить список всех информационных баз кластера с помощью метода `АдминистрированиеКластер.ПолучитьИнформационныеБазы()`, затем перебрать получившийся массив и по каким-то критериям выбрать одну или несколько информационных баз.
2. Сразу найти нужную информационную базы в том случае, если мы знаем ее уникальный идентификатор. Для получения описания конкретной информационной базы по известному идентификатору служит метод `АдминистрированиеКластер.ПолучитьИнформационнуюБазу()`.
3. Создать новую информационную базу с помощью функции `АдминистрированиеКластер.СоздатьИнформационнуюБазу()`. В этом случае необходимо указать все ключевые параметры информационной базы перед тем, произойдет фактическое создание базы. Фактическое создание базы будет выполнено после выполнения метода `АдминистрированиеИнформационнаяБаза.Записать()`.

Для того, чтобы выполнять какие-то административные действия, необходимо получить права администратора этой базы. Для этого следует выполнить аутентификацию от имени пользователя информационной базы (подробнее о пользователях написано здесь) системы "1С:Предприятие" с помощью метода `АдминистрированиеИнформационнаяБаза.ВыполнитьАутентификацию()`. После выполнения аутентификации появляется доступ к изменения свойств информационной базы.

Свойства информационной базы, которые доступны у объекта `АдминистрированиеИнформационнаяБаза`, по своим именам в основном соответствуют свойствам информационной базы, описанными в документации к системе "1С:Предприятие". Если свойства информационной базы изменены, то для того, чтобы обновить свойства информационной базы в кластере серверов, следует воспользоваться методом `АдминистрированиеИнформационнаяБаза.Записать()`.

Кроме изменения свойств информационной базы, система 1С:Исполнитель поддерживает следующие операции:

- Получить список текущих сеансов с помощью функции `АдминистрированиеИнформационнаяБаза.ПолучитьСеансы()`.
- Получить список соединений информационной базы с помощью функции`АдминистрированиеИнформационнаяБаза.ПолучитьСоединения()`.

- Удалить информационную базу из кластера серверов. Удаление выполняется с помощью метода `АдминистрированиеИнформационнаяБаза.Удалить()`. При этом режим удаления информационной базы задается с помощью параметра метода, который может принимать следующие значения:
  - `АдминистрированиеРежимУдаленияИнформационнойБазы.НеВыполнятьДействийСБазойДанных` - информационная база удаляется из кластера серверов, сама база данных не изменяется.
  - `АдминистрированиеРежимУдаленияИнформационнойБазы.ОчиститьБазуДанных` - информационная база удаляется из кластера серверов, база данных очищается.
  - `АдминистрированиеРежимУдаленияИнформационнойБазы.УдалитьБазуДанных` - информационная база удаляется из кластера серверов, база данных также удаляется.

Получить список информационных баз

```
метод Скрипт()
    пер ТекущийКластер = ПолучитьКластер()
    ТекущийКластер.ВыполнитьАутентификацию("", "")
    пер СписокИБ = ТекущийКластер.ПолучитьИнформационныеБазы()
    для текИБ из СписокИБ
        Консоль.Записать("ИБ, имя - " + текИБ.Имя)
        Консоль.Записать("\тСУБД: " + текИБ.Субд)
        Консоль.Записать("\тсервер баз данных: " + текИБ.СерверБазДанных)
        Консоль.Записать("\тимя базы данных: " + текИБ.ИмяБазыДанных)
        Консоль.Записать("\тописание: " + текИБ.Описание)
    ;
;

метод ПолучитьКластер(): АдминистрированиеКластер
    пер Сервер = новый АдминистрированиеСервера("localhost", 1545)
    возврат Сервер.ПолучитьКластеры()[0]
;
```

Сеансы и соединения

Сеансы и соединения - что это такое и что с ними можно сделать.

Общая информация

В системе "1С:Предприятие" под термином *сеанс* понимается описание какого-либо пользователя информационной базы и поток управления этого пользователя. Но сеанс не является средством доступа пользователя к кластеру серверов. В качестве средства доступа выступает *соединение*. При этом в каждый момент времени один сеанс обслуживается одним соединением. Допускается ситуация, что в разные моменты времени один сеанс обслуживается разными соединениями. Документация к системе "1С:Предприятие" содержит более подробную информацию о сеансах и соединениях.

Сеансы и работа с ними

Список сеансов можно получить как для всего кластера целиком, так и для конкретной информационной базы. В первом случае список сеансов будет содержать сеансы всех информационных баз кластера. Во втором случае в список попадут только сеансы конкретной информационной базы. В системе 1С:Исполнитель сеанс описывается объектом `АдминистрированиеСеанс`.

Доступ к данным сеансов можно получить разными способами:

- Для того, чтобы получить список всех сеансов кластера серверов, необходимо использовать функцию `АдминистрированиеКластер.ПолучитьСеансы()`.
- Для того, чтобы получить список сеансов конкретной информационной базы, следует использовать функцию `АдминистрированиеИнформаицоннаяБаза.ПолучитьСеансы()`.

- Для того, чтобы получить информацию о конкретном сеансе, можно использовать функцию АдминистрированиеКластер.ПолучитьСеанс(), однако для этого требуется знать уникальный идентификатор сеанса.

Свойства сеанса подробно описаны в документации по системе "1С:Предприятие". Кроме этого, стоит обратить внимание на следующие свойства объекта, которые позволяют получить объекты, описывающие все связанные объекты кластера серверов, обладая только информацией о сеансе:

- ИдентификаторИнформационнойБазы - с помощью функции АдминистрированиеКластер.ПолучитьИнформационнуюБазу() позволяет получить описание информационной базы.
- ИдентификаторПроцесса - с помощью функции АдминистрированиеКластер.ПолучитьРабочийПроцесс() позволяет получить описание рабочего процесса, который обслуживает текущий сеанс.
- ИдентификаторСеанса - тот самый уникальный идентификатор, используя который можно напрямую получить описание сеанса с помощью функции АдминистрированиеКластер.ПолучитьСеанс().
- ИдентификаторСоединения - позволяет с помощью функции АдминистрированиеКластер.ПолучитьСоединение() получить описание соединения, которое используется дол того, чтобы текущий сеанс каким-то образом взаимодействовал с кластером серверов.

С помощью объекта АдминистрированиеСеанс также предоставляется возможность прервать текущий северный вызов с помощью метода АдминистрированиеСеанс.ПрерватьТекущийСерверныйВызов(). Завершить сеанс, который описывается объектом АдминистрированиеСеанс, можно с помощью метода АдминистрированиеСеанс.ЗавершитьСеанс().

Соединения и работа с ними

Список соединений, как и список сеансов, может быть получен для всего кластера серверов или для одной информационный базы. В первом случае список соединений будет содержать соединения всех информационных баз кластера. Во втором случае в список попадут только соединения конкретной информационной базы. В системе 1С:Исполнитель соединение описывается объектом АдминистрированиеСоединение.

Доступ к данным соединений можно получить разными способами:

- Для того, чтобы получить список всех соединений кластера серверов, необходимо использовать функцию АдминистрированиеКластер.ПолучитьСоединения().
- Для того, чтобы получить список соединений конкретной информационной базы, следует использовать функцию АдминистрированиеИнформаиционнаяБаза.ПолучитьСоединения().
- Для того, чтобы получить информацию о конкретном соединении, можно использовать функцию АдминистрированиеКластер.ПолучитьСоединение(), однако для этого требуется знать уникальный идентификатор соединения.

Свойства соединения подробно описаны в документации по системе "1С:Предприятие". С помощью объекта АдминистрированиеСоединение предоставляется возможность разорвать соединение между клиентским приложением и кластером серверов. Для этого предназначен метод АдминистрированиеСоединение.Отключить().

Рабочие процессы

В разделе описано, как можно получить информацию о рабочем процессе.

Рабочий процесс - это основная "рабочая лошадка" кластера серверов. Именно рабочий процесс занимается обслуживанием клиентских соединений и взаимодействием с СУБД. Созданием и завершением работы рабочих процессов занимается *менеджер кластера* в соответствии с настройками кластера и текущей нагрузкой. Однако, инструменты администрирования кластера серверов позволяют получить список рабочих

процессов, работающих в данный момент времени. Для каждого из этих процессов можно получить их текущие параметры.

В системе 1С:Исполнитель рабочий процесс описывается объектом АдминистрированиеРабочийСервер. Для получения информации о рабочем процессе существует две возможности:

1. Получить список рабочих процессов кластера серверов с помощью функции АдминистрированиеКластера.ПолучитьРабочиеПроцессы(). Результатом работы функции будет массив объектов АдминистрированиеРабочийСервер. Затем можно выбрать требуемый рабочий процесс и получить его параметры.
2. Получить описание конкретного рабочего процесса с помощью функции АдминистрированиеКластера.ПолучитьРабочийПроцесс(). Для этого необходимо знать идентификатор требуемого рабочего процесса.

Параметры рабочего процесса подробно описаны в документации к системе "1С:Предприятие".
С точки зрения системы 1С:Исполнитель рекомендуется обратить внимание на свойство АдминистрированиеРабочийСервер.ИдентификаторРабочегоПроцесса, т.к. с помощью данного свойства можно получить описание рабочего процесса сразу, а не поиском в массиве рабочих процессов кластера серверов. Однако, следует учитывать, что в тот момент, когда вы захотите получить информацию о рабочем процессе, сам рабочий процесс может уже не существовать в системе.

Получить список рабочих процессов

```
метод Скрипт()
    пер мойКластер = ПолучитьКластер()
    мойКластер.ВыполнитьАутентификацию("", "")
    пер РабочиеПроцессы = мойКластер.ПолучитьРабочиеПроцессы()
    для Процесс из РабочиеПроцессы
        Консоль.Записать("Рабочий сервер - " + Процесс.ИмяКомпьютера + ":"
 + Процесс.Порт)
        Консоль.Записать("\тактивен: " + Процесс.Активен)
        Консоль.Записать("\твключен: " + Процесс.Включен)
        Консоль.Записать("\тзапущен: " + Процесс.ВремяЗапуска)
    ;
;

метод ПолучитьКластер(): АдминистрированиеКластер
    пер Сервер = новый АдминистрированиеСервера("localhost", 1545)
    возврат Сервер.ПолучитьКластеры()[0]
;
```

Профили безопасности

Описана работа с профилями безопасности.

Профили безопасности предназначены для того, чтобы ограничить использование различных внешних ресурсов кластером серверов. Такие ограничения обычно накладываются из соображений безопасности. Профиль безопасности описан в документации к системе "1С:Предприятие".

В системе 1С:Исполнитель профиль безопасности описывается объектом АдминистрированиеПрофильБезопасности. Получить список профилей безопасности, созданных в кластере серверов, можно с помощью метода АдминистрированиеКластер.ПолучитьПрофилиБезопасности().
Для создания нового профиля безопасности следует использовать метод АдминистрированиеКластер.СоздатьПрофильБезопасности().

Работу непосредственно с профилем безопасности можно условно разделить на две части: включение или выключение какой-либо настройки целиком и более точная настройка этой возможности. Для примера рассмотрим возможность работы с файловой системой. Свойство АдминистрированиеПрофильБезопасности.ПолныйДоступКФайловойСистеме управляет

доступном к файловой системе. Если это свойство установлено в значение Истина, то северная часть прикладного решения имеет полный доступ к файловой системе компьютера (-во) на котором работает кластер серверов.

Если свойство АдминистрированиеПрофильБезопасности.ПолныйДоступКФайловойСистеме установлено в значение Ложь, то полный доступ к файловой системе запрещен, но предоставляется возможность явно описать каталоги, куда прикладное решение может иметь доступ. Для этого необходимо использовать метод АдминистрированиеПрофильБезопасности.СоздатьРазрешенныйВиртуальныйКаталог(). Для получения текущего списка разрешенных каталогов используется метод АдминистрированиеПрофильБезопасности.ПолучитьРазрешенныеВиртуальныеКаталоги().

Таким образом, возможности работы с настраиваемыми возможностями профиля безопасности можно описать следующим образом:

- Работа с файловой системой.
  - Свойство: ПолныйДоступКФайловойСистеме.
  - Получить список: ПолучитьРазрешенныеВиртуальныеКаталоги().
  - Создать новый объект: СоздатьРазрешенныйВиртуальныйКаталог().
- Работа с COM-объектами. Только для кластера серверов, работающего под управлением ОС Windows.
  - Свойство: ПолныйДоступКCOMОбъектам.
  - Получить список: ПолучитьРазрешенныеCOMКлассы().
  - Создать новый объект: СоздатьРазрешенныйCOMКласс().
- Работа с внешними компонентами.
  - Свойство: ПолныйДоступКВнешнимКомпонентам.
  - Получить список: ПолучитьРазрешенныеВнешниеКомпоненты().
  - Создать новый объект: СоздатьРазрешеннуюВнешнююКомпоненту().
- Использование внешних модулей. Под "внешними модулями" понимаются внешние отчеты, внешние обработки, расширения и возможность использования оператора Выполнить() и функции Вычислить() (в терминологии системы "1С:Предприятие").
  - Свойство: ПолныйДоступКВнешнимМодулям.
  - Создать новый объект: СоздатьРазрешенныйВнешнийМодуль().
- Использование внешних приложений.
  - Свойство: ПолныйДоступКПриложениям.
  - Получить список: ПолучитьРазрешенныеВнешниеПриложения().
  - Создать новый объект: СоздатьРазрешенноеВнешнееПриложение().
- Использование ресурсов сети Интернет.
  - Свойство: ПолныйДоступКИнтернетРесурсам.
  - Получить список: ПолучитьРазрешенныеИнтернетРесурсы().
  - Создать новый объект: СоздатьРазрешенныйИнтернетРесурс().

Профиль безопасности, также, предоставляет возможность управлять следующими возможностями:

- Разрешить или запретить расширение прав доступа с помощью расширений конфигурации. Управляется свойством РазрешитьРасширениеПрав. Если свойство сброшено (установлено в значение Ложь), то система работает следующим образом: роль из расширения конфигурации может расширить право доступа только в том случае, если хотя бы в одной из ролей, перечисленных в свойстве профиля безопасности РолиОграничивающиеРасширениеПрав, установлено расширяемое право доступа. При этом необязательно, чтобы роли, перечисленные в свойстве профиля безопасности, использовались для пользователей информационной базы прикладного решения.

- Свойство РазрешитьУстановкуПривилегированногоРежима управляет возможностью включения привилегированного режима в том случае, если для профиля безопасности указана возможность использования в качестве профиля безопасности безопасного режима.
- Свойство ПрофильБезопасногоРежима, установленное в значение Истина, позволяют использовать такой профиль безопасности во встроенном языке системы "1С:Предприятие" и при подключении расширений конфигурации с помощью соответствующей стандартной функции.
- Свойство РазрешитьЗапускЛюбогоВнешнегоКодаВНебезопасномРежиме управляет возможностью запуска внешнего кода в небезопасном режиме. Если свойство сброшено, то прикладное решение может использовать только те модули, которые перечислены в свойстве

Глава 7

Механизмы работы с данными

Описание инструментов, которые используются для работы с различными форматами данных.

ПотокЧтения и ПотокЗаписи

Базовые типы для работы с различными форматами данных.

Поток - это логическое представление некоторого источника (или приемника) данных, например, файла на диске. С точки зрения потока, источник данных представляет собой последовательность байт. Данные можно читать из потока, и в этом случае можно говорить о потоке, предназначенном для чтения (и только для чтения). Такой поток в объектной модели представлен типом ПотокЧтения. Данные можно записывать в поток. В этом случае можно говорить о потоке, который предназначен только для записи. Такой поток в объектной модели представлен типом ПотокЗаписи.

Объекты типа ПотокЧтения и ПотокЗаписи являются транспортным уровнем для различных, более высокоуровневых, инструментов работы, например, работа с XML- или ZIP-файлами.

Объекты по работе с потоками являются потомками типа Закрываемое. Это значит, что переменные, в которых хранятся значения этих типов, можно объявлять с использованием модификатора исп, что обеспечит таким объектом таких типов гарантированное закрытие при выходе из области видимости. При закрытии потока записи происходит принудительная запись буфера в связанный приемник данных.

Для типов ПотокЧтения и ПотокЗаписи не заданы конструкторы. Объекты таких типов можно получить только от какого-то другого объекта, например, значения типа Файл. Закрытие объектов выполняется с помощь метода Закрыть(), при этом повторное закрытие потока не является ошибкой (метод является идемпотентным). Объект типа ПотокЗаписи использует буфер в оперативной памяти для оптимизации фактической записи.

Объекты работы с потоками предоставляют только базовый набор действий:

- ПотокЧтения позволяет копировать "свои" данные в поток записи (метод КопироватьВ()), а также позволяет прочитать поток как строку в кодировке UTF-8 (метод ПрочитатьКакТекст()).
- ПотокЗаписи позволяет принудительно записать на буфер в связанный приемник данных (метод СброситьБуферы()), а также записать в приемник данных строку в кодировке UTF-8 (метод Записать()).

ЧтениеДанных и ЗаписьДанных

Более высокоуровневые типы для работы с данными.

Общая информация

Типы ПотокЧтения и ПотокЗаписи обеспечивают самый минимальный набор методов для работы с данными. В некоторых случаях необходимо получить более высокоуровневые механизмы работы с данными, например, прочитать или записать число, найти какой-либо маркер в данных и т.д. Для выполнения таких действий предназначены объекты типа ЧтениеДанных и ЗаписьДанных. Рассмотрим эти объекты более подробно.

Запись данных

Для записи данных предназначен объект ЗаписьДанных. Он создается конструктором, параметрами которого выступают поток, в который будут записываться данные, и настройки записи данных. Настройки записи данных задаются с помощью одноименного объекта (тип НастройкиЗаписиДанных), который позволяет управлять следующими настройками:

- В какой кодировке будут записываться данные в данном потоке (если в приемник данных пишется текстовая информация). Задается текстовым идентификатором кодировки. Свойство Кодировка.
- Позволяет указать, каким образом в приемник данных будет записываться метка порядка байтов (также известная как BOM (Byte Order Mark). Данная настройка имеет учитывается только в том случае, если для потока устанавливается кодировка UTF-8, UTF-16 и UTF-32. Для остальных кодировок данная

настройка игнорируется. Задается с помощью значения типа МеткаПорядкаБайтов. Свойство МеткаПорядкаБайтов.

- Какой символ или их последовательность будет считаться разделителем строк (если в приемник данных пишется текстовая информация). Является строкой. Свойство РазделительСтрок.
- Позволяет указать порядка следования байтов в словах: «младший-старший» (little endian) или «старший-младший» (big endian). Порядок следования байтов важен при записи многобайтовых данных (слов). Этот порядок может определяться архитектурой компьютера, на котором будут читаться данные, записываемые в поток, или этот порядок определяется самим форматом записываемых данных (например, форматом файла). Задается с помощью значения типа ПорядокБайтов. Свойство ПорядокБайтов.

Объект ЗаписьДанных позволяет выполнять следующие операции:

- Запись одного байта (числа в интервале от 0 до 255 включительно). Используется метод ЗаписатьБайт().
- Записать целое число. Поддерживаются 16-разрядные, 32-разрядные и 64-разрядные целые числа. При записи можно указать, с каким порядком байт будет записывать число. Если порядок не указан - будет использован порядок байт из настроек объекта ЗаписьДанных. Для записи числе используются методы ЗаписатьЦелое16(), ЗаписатьЦелое32() и ЗаписатьЦелое64().
- Записать несколько символов или строку символов. Используются методы ЗаписатьСимволы() и ЗаписатьСтроку(). Разница заключается в том, что метод ЗаписатьСтроку() после самой строки записывает в поток еще и разделитель строк. Используется или разделитель строк, указанный в параметре метода или в настройках объекта ЗаписьДанных. Также в обеих методах имеется возможность указать кодировку записываемых данных. Если кодировка не указана в методе - будет использоваться кодировка из настроек объекта ЗаписьДанных.
- Записать в поток двоичные данные. Двоичные данные могут быть представлены или типом Байты или объектом типа РезультатЧтенияДанных, который получается с помощью интерфейса объекта ЧтениеДанных.

Чтение данных

Для чтения данных предназначен объект ЧтениеДанных. Он создается конструктором, параметрами которого выступают поток, из которого будут читаться данные (источник данных), и настройки чтения данных. Настройки чтения данных задаются с помощью одноименного объекта (тип НастройкиЧтенияДанных), который позволяет управлять следующими настройками:

- В какой кодировке будут читаться данные из данного потока (если из источника данных читается текстовая информация). Задается текстовым идентификатором кодировки. Свойство Кодировка.
- Какой символ или их последовательность будет считаться разделителем строк (если из источника данных читается текстовая информация). Может быть простой строкой или массивом строк (если возможно использование нескольких разделителей строк). Свойство РазделителиСтрок.
- Позволяет указать порядка следования байтов в словах: «младший-старший» (little endian) или «старший-младший» (big endian). Порядок следования байтов важен при записи многобайтовых данных (слов). Этот порядок может определяться архитектурой компьютера, на котором записывались данные, которые будут считываться из потока, или этот порядок определяется самим форматом читаемых данных (например, форматом файла). Задается с помощью значения типа ПорядокБайтов. Свойство ПорядокБайтов.

Объект ЧтениеДанных позволяет выполнять следующие операции:

- Проверить, что в источнике данных есть данные. Если при очередном чтении было прочитано меньше данных, чем было запрошено, возвращает значение Истина.
- Прочитать один байт. Используется метод ПрочитатьБайт().
- Прочитать целое число. Поддерживаются 16-разрядные, 32-разрядные и 64-разрядные целые числа. При записи можно указать, с каким порядком байт будет считываться число. Если порядок не указан - будет использован порядок байт из настроек объекта ЧтениеДанных. Для записи числе используются методы ПрочитатьЦелое16(), ПрочитатьЦелое32() и ПрочитатьЦелое64().

- Прочитать несколько символов или строку символов. Используются методы `ПрочитатьСимволы()` и `ПрочитатьСтроку()`. Разница заключается в том, что метод `ПрочитатьСтроку()` считывает поток до ближайшего разделителя строк, который указывается в качестве параметра метода или получается из настроек объекта `ЧтениеДанных`. Метод `ПрочитатьСимволы()` считает ровно то количество символов, которое указаны в параметре метода. В обеих методах имеется возможность указать кодировку считываемых данных. Если кодировка не указана в методе - будет использоваться кодировка из настроек объекта `ЧтениеДанных`.
- Передвинуть позицию чтения в потоке вперед на какое-то количество байт (с помощью метода `Пропустить()`) или до выбранного маркера (метод `ПропуститьДо()`). Пр пропуске байт указывается - сколько байт надо пропустить. При пропуске до маркера - указывается один (в виде строки) или несколько (в виде массива) маркеров, по достижению которого (или одного из которых) пропуск прекращается. Для маркера можно указать используемую кодировку.
- Прочитать массив данных произвольного размера (с помощью метода `Прочитать()`). Метод вернет объект типа `РезультатЧтенияДанных`, который предназначен для хранения и обработки прочитанных данных. Метод `ПрочитатьДо()` отличается от метода `Прочитать()` только тем, что метод `ЧтениеДо()` позволяет прочитать данные, размер которых заранее неизвестен. Но известен маркер, которым заканчивается текущая порция (или начинается следующая).

Результат чтения данных

Объект `РезультатЧтенияДанных` получается при чтении данных из потока. Этот тип также может использоваться для записи данных в поток. Данный тип является закрываемым.

После получения значения данного типа, можно получить размер считанных данных (свойство `Размер`). Если при чтении данных использовались маркеры, то свойства `МаркерНайден` и `ИндексМаркера`, позволят узнать, что чтение было остановлено в результате нахождения маркера, а также понять, какой маркер послужил был обнаружен.

Полившиеся данные можно использовать как источник для получения значения типа `Байты` (метод `ПолучитьБайты()`) или `ПотокЧтения` (метод `ОткрытьПотокДляЧтения()`). Каждый из этих методов можно вызывать многократно. До момента закрытия объекта, методы будут возвращать одинаковый результат.

Данные, которые считаны в объект `РезультатЧтенияДанных`, могут быть сохранены во временном файле на диске. Файл удаляется после вызова метода объекта `Закрыть()` (явного или неявного).

Работа с **XML**

Описывается работа с документами в формате XML.

Что такое **XML?**

XML (e**X**tensible **M**arkup **L**anguage) - это расширяемый язык разметки. Данные, которые представлены в формате XML, представляют из себя текстовые данные, которые помещаются в *элементы*. Элементы, в свою очередь, образуют *документ*. Элемент представляет собой данные, окруженные *тегами*. Данные, в свою очередь, также могут быть элементами (т.е. поддерживается вложенная структура элементов). Тег может быть открывающий и закрывающий. Тег всегда обрамлен символами "<" и ">" (угловые скобки). Открывающий тег содержит только имя тега в окружении угловых скобок. Закрывающий тег отличается от открывающего тем, что после открывающей угловой скобки и до имени тега, размещается признак закрывающего тега - символ "/". В открывающем теге могут находится один или несколько *атрибутов*. Атрибут - это пара ключ-значение: `key="value"`. Атрибуты разделяются пробелами. В документе XML может быть только один тег. Этот тег называется *корневым тегом*. Вышеприведенное описание не является исчерпывающим. Оно призвано дать общее описание формата, работа с которым будет описана в данном разделе. Формальное и подробное описание формата XML можно найти здесь.

Пример **XML**-файла

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<note>
    <to>Иван</to>
    <from>Петр</from>
    <heading>Напоминание</heading>
    <body>Не забудь о встрече в эти выходные!</body>
</note>
```

Способы работы с **XML**

Для работы с XML можно использовать одну из двух техник: потоковая чтение/запись документа XML и использование объектной модели документа (DOM-модель). Каждая из указанных техник имеет свои плюсы и минусы. Данный раздел описывает работу исключительно в потоковой технике.

Запись **XML**

В данном разделе приводится краткая информация о том, как записать XML-документ.

Запись простого **XML**-документа

Для записи XML-документа предназначен объект `ЗаписьXml`. В общем случае, записывать XML-документ можно в объект, производный от объекта `ПотокЗаписи`. Мы будем рассматривать запись документа в файл.

Самый простой случай XML-документа - это объявление документа и корневой тег. В нашем случае это тег `<note>`. Объявление документа - это обязательный элемент любого XML-документа. Поэтому использование пары методов `ЗаписатьНачалоДокумента()`/`ЗаписатьКонецДокумента()` является обязательным для записи любого XML-документа. Таким же обязательным элементом любого XML-документа является корневой тег.

При работе с XML-документ важно запомнить еще один момент: программист записывает не только открывающий тег, но и закрывающий! В противном случае структура XML-документа будет нарушена. Именно поэтому многие методы работы с XML используются парами. Например, для записи какого-либо элемента используется два метода:

- `ЗаписатьНачалоЭлемента()` - для того, чтобы записать открывающий тег: `<note>`.
- `ЗаписатьКонецЭлемента()` - для того, чтобы записать закрывающий тег: `</note>`.

Запись **XML.** Шаг **1**

```
метод Скрипт()
    пер Файл = новый Файл("c:\\temp\\note.xml")
    пер Запись = новый ЗаписьXml(Файл.ОткрытьПотокЗаписи())
    Запись.ЗаписатьНачалоДокумента()
    Запись.ЗаписатьНачалоЭлемента("note")
    Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецДокумента()
;
```

Запись элементов **XML**-документа

После того, как мы сформировали простой XML-документ, наполним этот документ оставшимся содержимым. Т.к. XML-документ имеет иерархическую структуру, где иерархию образуют вложенные теги, то записывать теги нам будет нужно с учетом их вложенности. Для этого необходимо начинать починенный элемент после указания начала родительского элемента, но **до(!)** окончания родительского элемента.

Каждый тег записывается своей парой методов `ЗаписатьНачалоЭлемента()`/`ЗаписатьКонецЭлемента()`. В результате работы нашего примера получится документ, очень похожий на оригинальный, только в нем отсутствуют значения элементов.

```
<?xml version='1.0' encoding='UTF-8'?>
<note>
```

```
  <to/>
  <from/>
  <heading/>
  <body/>
</note>
```

В исходном тексте, который формирует новый XML-документ, отступ операторов, которые формируют элементы `<to>`, `<from>`, `<heading>` и `<body>`, сделан исключительно из соображений более удобного восприятия примера. В реальных программах так делать не обязательно.

Запись **XML.** Шаг **2**

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\note.xml")
    пер Запись = новый ЗаписьXml(Файл.ОткрытьПотокЗаписи())
    Запись.ЗаписатьНачалоДокумента()
    Запись.ЗаписатьНачалоЭлемента("note")
        Запись.ЗаписатьНачалоЭлемента("to")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("from")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("heading")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("body")
        Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецДокумента()
;
```

Запись содержимого элементов **XML-**документа

Для записи текста, который будет расположен "внутри" тегов XML-документа, предназначен метод `ЗаписатьТекст()`. параметр этого метода будет записан в качестве содержимого того элемента, который в данный момент является открытым. Теперь наш документ стал полностью соответствовать примеру.

```
<?xml version='1.0' encoding='UTF-8'?><note>
  <to>Иван</to>
  <from>Петр</from>
  <heading>Напоминание</heading>
  <body>Не забудь о встрече в эти выходные</body>
</note>
```

Запись **XML.** Шаг **3**

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\note.xml")
    пер Запись = новый ЗаписьXml(Файл.ОткрытьПотокЗаписи())
    Запись.ЗаписатьНачалоДокумента()
    Запись.ЗаписатьНачалоЭлемента("note")
        Запись.ЗаписатьНачалоЭлемента("to")
            Запись.ЗаписатьТекст("Иван")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("from")
            Запись.ЗаписатьТекст("Петр")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("heading")
            Запись.ЗаписатьТекст("Напоминание")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("body")
```

```
            Запись.ЗаписатьТекст("Не забудь о встрече в эти выходные")
        Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецДокумента()
;
```

Запись атрибутов элемента **XML**

Последним действием, которое мы рассмотрим в части записи XML-документа - запись атрибута XML-элемента. Допустим, мы хотим добавить к отправителю и получателю нашей заметки псевдонимы отправителя и получателя. Для этого будем использовать атрибут `@nickname`.

Для записи атрибута используется метод `ЗаписатьАтрибут()`. Записывать атрибут необходимо сразу после метода `ЗаписатьНачалоЭлемента()`. В метод передаются сразу оба значения: имя атрибута и его (атрибута) значение. В результате наш XML-документ будет выглядеть следующим образом:

```
<?xml version='1.0' encoding='UTF-8'?>
<note>
  <to nickname="developer">Иван</to>
  <from nickname="boss">Петр</from>
  <heading>Напоминание</heading>
  <body>Не забудь о встрече в эти выходные</body>
</note>
```

Если требуется указать несколько атрибутов для одного элемента - значит надо использовать метод `ЗаписатьАтрибут()` столько раз, сколько атрибутов требуется указать.

Запись **XML.** Шаг **4**

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\note.xml")
    пер Запись = новый ЗаписьXml(Файл.ОткрытьПотокЗаписи())
    Запись.ЗаписатьНачалоДокумента()
    Запись.ЗаписатьНачалоЭлемента("note")
        Запись.ЗаписатьНачалоЭлемента("to")
            Запись.ЗаписатьАтрибут("nickname", "developer")
            Запись.ЗаписатьТекст("Иван")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("from")
            Запись.ЗаписатьАтрибут("nickname", "boss")
            Запись.ЗаписатьТекст("Петр")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("heading")
            Запись.ЗаписатьТекст("Напоминание")
        Запись.ЗаписатьКонецЭлемента()
        Запись.ЗаписатьНачалоЭлемента("body")
            Запись.ЗаписатьТекст("Не забудь о встрече в эти выходные")
        Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецЭлемента()
    Запись.ЗаписатьКонецДокумента()
;
```

Чтение **XML**

В данном разделе приводится краткая информация о том, как прочитать XML-документ.

Перебор элементов **XML**-документа

Для чтения XML-документа предназначен объект `ЧтениеXml`. В общем случае, читать XML-документ можно из объекта, производного от объекта `ПотокЧтения`. Мы будем рассматривать чтение документа из

файла. В качестве примера файла рассмотрим фрагмент манифеста универсального приложения ОС Windows. Это достаточно сложный XML-документ, который позволит увидеть основные особенности чтения таких документов.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Package IgnorableNamespaces="uap mp build"
 xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
 xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
 xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10"
 xmlns:build="http://schemas.microsoft.com/developer/appx/2015/build">

  <mp:PhoneIdentity
  PhoneProductId="A588A326-FD4F-441C-83B2-AA0B8554C548"
  PhonePublisherId="00000000-0000-0000-0000-000000000000"/>
  <Properties>
    <DisplayName>ms-resource:AppName</DisplayName>
    <PublisherDisplayName>1C LLC</PublisherDisplayName>
    <Logo>icon_50x50.png</Logo>
  </Properties>
  <Resources>
    <Resource Language="EN-US"/>
  </Resources>
  <Capabilities>
    <uap:Capability Name="appointments"/>
    <uap:Capability Name="contacts"/>
    <Capability Name="internetClient"/>
    <Capability Name="privateNetworkClientServer"/>
    <Capability Name="internetClientServer"/>
    <uap:Capability Name="musicLibrary"/>
    <uap:Capability Name="picturesLibrary"/>
    <uap:Capability Name="removableStorage"/>
    <uap:Capability Name="videosLibrary"/>
    <DeviceCapability Name="location"/>
    <DeviceCapability Name="webcam"/>
    <DeviceCapability Name="microphone"/>
    <DeviceCapability Name="proximity"/>
  </Capabilities>
  <build:Metadata>
    <build:Item Name="cl.exe" Version="19.16.27030.1 built by: vcwrkspc"/>
    <build:Item Name="VisualStudio" Version="15.0"/>
    <build:Item Name="OperatingSystem" Version="6.3.9600.16384
 (winblue_rtm.130821-1623)"/>
    <build:Item Name="Microsoft.Build.AppxPackage.dll"
 Version="15.0.28307.104"/>
    <build:Item Name="ProjectGUID" Value="{15630E4C-5D90-44AB-9CF0-
FBC27700DDAA}"/>
    <build:Item Name="OptimizingToolset" Value="None"/>
    <build:Item Name="TargetRuntime" Value="Native"/>
    <build:Item Name="Microsoft.Windows.UI.Xaml.Build.Tasks.dll"
 Version="15.0.28307.102"/>
    <build:Item Name="WindowsMobile" Version="10.0.17763.0"/>
    <build:Item Name="MakePri.exe" Version="10.0.17763.132
 (WinBuild.160101.0800)"/>
  </build:Metadata>
</Package>
```

Рассматриваемый объект читает XML-документ строго последовательно. Каждый элемент имеет свое имя, которое отображается в свойство `Имя` объекта `ЧтениеXml`. Простейший пример такого чтения будет выглядеть примерно следующим образом:

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
```

```
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        Консоль.Записать("Имя узла - " + Чтение.Имя)
    ;
;
```

Вызов метода `Следующий()` приводит к последовательному обходу всех элементов читаемого XML-документа. Когда данный метод вернет значение `Ложь`, это будет означать, что файл завершился. Результат исполнения этого программного кода будет не очень понятным (приведен фрагмент):

```
Имя узла - Package
Имя узла - mp:PhoneIdentity
Имя узла - mp:PhoneIdentity
Имя узла - Properties
Имя узла - DisplayName
Имя узла -
Имя узла - DisplayName
Имя узла - PublisherDisplayName
Имя узла -
Имя узла - PublisherDisplayName
Имя узла - Logo
Имя узла -
Имя узла - Logo
Имя узла - Properties
...
```

Первое, что смущает при взгляде на результат работы программы - некоторые элементы выводятся один раз, некоторые - два раза, а некоторые элементы вообще не имеют имени. Попробуем разобраться с этим вопросом.

Каждый элемент XML имеет открывающий и закрывающий тег. При этом оба этих тега в теле документа имеют одинаковое имя: `<tag>`текст`</tag>`. Закрывающий тег несколько отличается, но это нам сейчас не интересно. Т.е. объект `ЧтениеXml` читает наш документ последовательно, а при чтении структура документа должна быть полностью восстановлена, то при чтении отдельно читается начало элемента и отдельно - его окончание. А так как имена в открывающем и закрывающем тегах одинаковые - мы видим две строки с одним именем. Соответственно, если строка с именем только одна - скорее всего, до закрывающего тега еще не дошли.

Чтобы определить, какой элемент в данный момент считан из нашего документа, объект `ЧтениеXml` предоставляет свойство `ТипУзла`. С помощью этого свойства мы может проверить наше предыдущее утверждение:

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        Консоль.Записать("Имя узла - " + Чтение.Имя + ", тип - "
 + Чтение.ВидУзла)
    ;
;
```

Результат работы нашего примера будет следующим (приведен фрагмент того же размера):

```
Имя узла - Package, тип - StartElement
Имя узла - mp:PhoneIdentity, тип - StartElement
Имя узла - mp:PhoneIdentity, тип - EndElement
Имя узла - Properties, тип - StartElement
Имя узла - DisplayName, тип - StartElement
Имя узла - , тип - Text
```

```
Имя узла - DisplayName, тип - EndElement
Имя узла - PublisherDisplayName, тип - StartElement
Имя узла - , тип - Text
Имя узла - PublisherDisplayName, тип - EndElement
Имя узла - Logo, тип - StartElement
Имя узла - , тип - Text
Имя узла - Logo, тип - EndElement
Имя узла - Properties, тип - EndElement
...
```

Посмотрим на результат работы нашей программы. Мы видим, что тип элемента (или *узла* XML-документа) в каждой строке вывода изменяется. В текущем примере мы видим три разных значения:

- *StartElement* - таким образом описывается начало элемента.
- *EndElement* - таким образом описывается окончание элемента.
- *Text* - таким типом отмечается содержимое узла XML-документа, если это содержимое не является другим элементом.

Мы видим, что наши утверждения, высказанные ранее, подтвердились практически. Также получили объяснения элементы, которые не имеют имени. В XML-документе узлы, которые имеют содержимое, выглядят следующим образом: `<tag>содержимое</tag>`. В то же время, не все узлы такое содержимое имеют. Для того, чтобы определить, имеет узел содержимое или нет, предназначено свойство `ЧтениеXml.ИмеетЗначение`.

Наш пример, модифицированный для того, чтобы выводить содержимое для элементов, которые его содержат, выглядит следующим образом:

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        Консоль.Записать("Имя узла - " + Чтение.Имя + ", тип - "
 + Чтение.ВидУзла)
        если Чтение.ИмеетЗначение
            Консоль.Записать("Содержимое: " + Чтение.Значение)
        ;
    ;
;
```

Результатом работы этой программы будет следующий:

```
Имя узла - Package, тип - StartElement
Имя узла - mp:PhoneIdentity, тип - StartElement
Имя узла - mp:PhoneIdentity, тип - EndElement
Имя узла - Properties, тип - StartElement
Имя узла - DisplayName, тип - StartElement
Имя узла - , тип - Text
Содержимое: ms-resource:AppName
Имя узла - DisplayName, тип - EndElement
Имя узла - PublisherDisplayName, тип - StartElement
Имя узла - , тип - Text
Содержимое: 1C LLC
Имя узла - PublisherDisplayName, тип - EndElement
Имя узла - Logo, тип - StartElement
Имя узла - , тип - Text
Содержимое: icon_50x50.png
Имя узла - Logo, тип - EndElement
Имя узла - Properties, тип - EndElement
...
```

Кроме безусловного чтения следующего элемента, с помощью объекта `ЧтениеXml` можно читать элементы какого-то заранее известного имени. Чтение будет выполняться без учета иерархии элементов. Для того, чтобы выполнить такое чтение, необходимо использовать метод `СледующийДо()`. Параметром метода является имя элемента, начало которого необходимо прочитать. После того, как выполнено позиционирование на требуемый элемент, необходимо вызвать метод `Следующий()`, а затем снова вызвать метод `СледующийДо()`. В примере будет демонстрироваться чтение элемента с именем *Capability*, которых в нашем примере находится три элемента.

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.СледующийДо("Capability")
        Консоль.Записать("Имя узла - " + Чтение.Имя + ", тип - "
 + Чтение.ВидУзла)
        если Чтение.ИмеетЗначение
            Консоль.Записать("Содержимое: " + Чтение.Значение)
        ;
        Чтение.Следующий()
    ;
;
```

В том случае, если какой-либо узел нам не нужен, мы можем пропустить все его (узла) содержимое и возобновить чтение с узла, который следует в нашем XML-документ **после** пропускаемого узла. Узел пропускается полностью, включая все вложенные узлы, если таковые имеются. Для того, чтобы пропустить узел, необходимо использовать метод `Пропустить()`. В следующем примере мы дойдем до узла `<Capabilities>`, затем пропустим его и сразу окажем на стартовом элементе узла `<build:Metadata>`:

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        если Чтение.ВидУзла == ВидУзлаXml.НачалоЭлемента
            Консоль.Записать("Имя узла - " + Чтение.Имя + ", тип - "
 + Чтение.ВидУзла)
            если Чтение.Имя == "Capabilities"
                Консоль.Записать("Пропускаем этот узел!")
                Чтение.Пропустить()
            ;
        ;
    ;
;
```

Также при переборе элементов полезным может оказаться метод `ЭтоПустойЭлемент()`, который позволяет понять, что у текущего узла нет значения. Другими словами, метод `ЭтоПустойЭлемент()` вернет значение `Истина` для узла вида `<tag/>`.

Чтение атрибутов узла **XML-**документа

Узлы XML-документа могут иметь один или несколько атрибутов. Зачастую смысловая информация узла расположена именно в атрибутах, а не в содержимом элемента. Для того, чтобы работать с атрибутами узла XML, встроенный язык предоставляет несколько методов, среди которых:

- `КоличествоАтрибутов()` - позволяет получить количество атрибутов у данного элемента.
- `ИмяАтрибута()` - позволяет получить имя атрибута по индексу.
- `ЗначениеАтрибута()` - позволяет получить значение атрибута по имени атрибута.
- `ЗначениеАтрибутаПоИндексу()` - позволяет получить значение атрибута по индексу атрибута.

Работать с атрибутами можно только для такого узла, который является началом элемента. Расширим пример чтения XML-документа кодом, который будет отображать атрибуты каждого узла (если атрибуты присутствуют). Код будет иметь следующий вид:

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        Консоль.Записать("Имя узла - " + Чтение.Имя + ", тип - "
 + Чтение.ВидУзла)
        если Чтение.ИмеетЗначение
            Консоль.Записать("Содержимое: " + Чтение.Значение)
        ;
        если Чтение.ВидУзла != ВидУзлаXml.НачалоЭлемента
            продолжить
        ;
        для индекс = 0 по Чтение.КоличествоАтрибутов()-1
            пер ИмяАтрибута = Чтение.ИмяАтрибута(индекс)
            Консоль.Записать("\тАтрибут: " + ИмяАтрибута + " = "
 + Чтение.ЗначениеАтрибута(ИмяАтрибута))
        ;
    ;
;
```

Результат работы этого примера будет следующим:

```
Имя узла - Package, тип - StartElement
 Атрибут: IgnorableNamespaces = uap mp build
Имя узла - mp:PhoneIdentity, тип - StartElement
 Атрибут: PhoneProductId = A588A326-FD4F-441C-83B2-AA0B8554C548
 Атрибут: PhonePublisherId = 00000000-0000-0000-0000-000000000000
Имя узла - mp:PhoneIdentity, тип - EndElement
Имя узла - Properties, тип - StartElement
Имя узла - DisplayName, тип - StartElement
Имя узла - , тип - Text
Содержимое: ms-resource:AppName
Имя узла - DisplayName, тип - EndElement
Имя узла - PublisherDisplayName, тип - StartElement
Имя узла - , тип - Text
Содержимое: 1C LLC
Имя узла - PublisherDisplayName, тип - EndElement
Имя узла - Logo, тип - StartElement
Имя узла - , тип - Text
Содержимое: icon_50x50.png
Имя узла - Logo, тип - EndElement
Имя узла - Properties, тип - EndElement
...
```

Чтение содержимого как данные различных типов

Для чтения значения элемента предназначены свойства объекта `ЧтениеXml` `ИмеетЗначение` и `Значение`. Первое свойство позволяет нам узнать, что элемент имеет значение, а второе свойство предоставляет доступ к этому значению. Однако, значение получается исключительно в текстовом виде. Это не всегда бывает удобно. Частно нам точно известно, что значение того или иного элемента XML-документа имеет конкретный тип. Объект `ЧтениеXml` предоставляет нам несколько методов, которые помогают считывать значения некоторых типов. Все эти методы начинаются с префикса `ПрочитатьСодержимоеКак`:

- `ПрочитатьСодержимоеКакБулево()`. Данный метод считывает значение узла и преобразует его к типу `Булево`. Текст *true* и *1* преобразуются в значение `Истина`, а текст *false* и *0* преобразуются в значение `Ложь`.

- ПрочитатьСодержимоеКакЧисло(). Данный метод преобразует содержимое элемента к типу Число. При этом игнорируются начальные и конечные пробельные символы.
- ПрочитатьСодержимоеКакУуид(). Данный метод считывает содержимое и считает, что оно является уникальным идентификатором (*GUID*). Возвращается значение типа Ууид. При чтении игнорируются начальные и конечные пробельные символы.

Пространства имен

Имена элементов и атрибутов в XML-документ могут быть квалифицированными и неквалифицированными (локальными). Квалифицированное имя уникально в рамках XML-документа. Квалифицированное имя состоит из префикса, определяющего пространство имен, и локального имени. Префикс и локальное имя разделяются символом двоеточия (":"). Когда XML-документ читается с помощью объекта ЧтениеXml, то программист может получить доступ к любой части имени элемента или атрибута:

- Для элемента:
  - Свойство Имя. Содержит квалифицированное имя текущего узла. Квалифицированное имя состоит из префикса и локального имени (разделенного символом двоеточия).
  - Свойство Префикс. Содержит префикс пространства имен для текущего узла.
  - Свойство ЛокальноеИмя. Содержит локальное имя текущего узла документа.
  - Свойство URIПространстваИмен. Данное свойство позволяет получить URI пространства имен. Префикс этого пространства имен можно получить с помощью свойства Префикс.
- Для атрибута:
  - Метод ИмяАтрибута(). Возвращает квалифицированное имя текущего атрибута. Квалифицированное имя состоит из префикса и локального имени (разделенного символом двоеточия).
  - Метод ПрефиксАтрибута(). Содержит префикс пространства имен для текущего атрибута.
  - Свойство ЛокальноеИмяАтрибута(). Содержит локальное имя текущего атрибута элемента.
  - Свойство URIПространстваИменАтрибута(). Данное свойство позволяет получить URI пространства имен атрибута. Префикс пространства имен атрибута можно получить с помощью метода ПрефиксАтрибута().

Далее приведен пример программного кода, который отображает всю информацию об узлах и атрибутах XML-документа, связанную с пространствами имен. Будет выведено квалифицированное имя, префикс имени, локальное имя и URI пространства имен.

```
метод Скрипт()
    пер ВременныйКаталог = СредаИсполнения.ПолучитьПеременную("temp")
    пер Файл = новый Файл(ВременныйКаталог + "\\manifest.xml")
    пер Чтение = новый ЧтениеXml(Файл.ОткрытьПотокЧтения())
    пока Чтение.Следующий()
        если Чтение.ВидУзла == ВидУзлаXml.НачалоЭлемента
            Консоль.Записать("Имя узла: " + Чтение.Имя)
            Консоль.Записать("\tПрефикс              : " + Чтение.Префикс)
            Консоль.Записать("\tЛокальное имя        : "
 + Чтение.ЛокальноеИмя)
            Консоль.Записать("\tURI пространства имен: "
 + Чтение.ПространствоИмен)
            для индекс = 0 по Чтение.КоличествоАтрибутов()-1
                пер ИмяАтрибута = Чтение.ИмяАтрибута(индекс)
                Консоль.Записать("\т\тИмя атрибута: " + ИмяАтрибута)
                Консоль.Записать("\т\т\тПрефикс              : "
 + Чтение.ПрефиксАтрибута(индекс))
                Консоль.Записать("\т\т\тЛокальное имя        : "
 + Чтение.ЛокальноеИмяАтрибута(индекс))
                Консоль.Записать("\т\т\тURI пространства имен: "
 + Чтение.ПространствоИменАтрибута(индекс))
            ;
        ;
```

```
    ;
;
```

Работа с **zip**-архивом

В данном разделе будут рассмотрены типы, которые используются при работе с zip-архивом.

ZIP - это популярный формат сжатия данных без потерь. Этот формат часто используется для создания файловых *архивов*: файлов специального формата, которые содержат другие файлы, сжатые с использованием алгоритма ZIP.

Для записи архива используется объект `ЗаписьZip`. Архив будет записываться в какой-либо поток, предназначенный для записи. Такой поток, например, можно получить из значения типа `Файл` следующим образом: `Файл.ОткрытьПотокЗаписи()`. Так же для архива можно указать пароль, уровень сжатия и комментарий. Для добавления файла в создаваемый архив предназначен метод `Добавить()` типа `ЗаписьZip`. При добавлении файла в архив следует помнить о том, как в архиве формируется иерархия файлов. Параметр `ПутьВАрхиве` метода `Добавить()` позволяет указать путь к добавляемому файлу. Если указать в этом параметр полный путь к добавляемому файлу - в архиве будет сохранен именно он. Если требуется указать относительный путь к файл - требуется самостоятельно удалить фрагмент пути от корня файловой системы до нужного уровня.

Далее будет приведен пример добавления в архив всех файлов с расширением `.xml` в каталоге `Documents` домашнего каталога текущего пользователя (с обходом всех подкаталогов). Файлы будут сохраняться в архиве по относительным путям. Корневым каталогом архива будет каталог `Documents`.

```
метод Скрипт()
    пер Источник = новый Файл(Файлы.ПолучитьДомашнийКаталог().Путь + "\
\Documents")
    пер ИмяАрхива = новый Файл(Источник.Путь + "\\xml-files.zip")
    пер Писатель = новый ЗаписьZip(ИмяАрхива.ОткрытьПотокЗаписи())
    для Файл из Источник.Дочерние
        ДобавитьРекурсивно(Писатель, Файл, ИмяАрхива.Каталог.Путь)
    ;
    Писатель.Записать()
    Консоль.Записать("Архивация завершена.")
;

метод ДобавитьРекурсивно(ПотокАрхива: ЗаписьZip, Источник: Файл, КорневойКаталог: Строка
    если Источник.ЯвляетсяФайлом() и Источник.Расширение == "xml"

  ПотокАрхива.Добавить(Источник.ОткрытьПотокЧтения(), Источник.Путь.Заменить(КорневойКа
 ""))
    ;
    для Файл из Источник.Дочерние
        ДобавитьРекурсивно(ПотокАрхива, Файл, КорневойКаталог)
    ;
;
```

В данном примере формирование относительного пути в архиве происходит при выполнении следующей операции: `Источник.Путь.Заменить(КорневойКаталог, "")`.

Для чтения ZIP-архива используется объект `ЧтениеZip`. Общая схема работы с данным объектом подобна записи архива: для файла с архивом получается поток чтения; затем создается объект `ЧтениеZip`, который читает данные из потока чтения; затем происходит получение содержимого архива. Для обхода содержимого архива используется пара методов: `Следующий()` - который позволяет понять, есть еще файл в архиве, и `ПолучитьЭлемент()` - который получает описание файла, который стал текущим после выполнения метода `Следующий()`.

Следующий пример ищет в каталоге (который задается переменной *ГдеИщем*) все ZIP-архивы и выводит в консоль список файлов для каждого найденного архива.

```
метод Скрипт()
    пер ГдеИщем = "<путь к каталогу с zip-файлами>"
    пер СписокФайлов
 = Файлы.Найти(ГдеИщем, новый НастройкиПоискаФайлов().ИмяСодержит(".zip"))
    для Файл из СписокФайлов
        пер Архив = новый ЧтениеZip(Файл.ОткрытьПотокЧтения())
        Консоль.Записать("ZIP-файл: " + Файл.Имя)
        пока Архив.Следующий()
            пер Элемент = Архив.ПолучитьЭлемент()
            Консоль.Записать("\tэлемент: " +  Элемент.ПутьВАрхиве +
 (Элемент.ЯвляетсяКаталогом() ? " (каталог)" : ""))
        ;
    ;
;
```