

ТУРБО 9

Руководство программиста

Москва
2008

Турбо 9: Руководство программиста. М.: ДИЦ, 2008.

Программное обеспечение и настоящий документ не могут быть скопированы, размножены, использованы по частям для составления других текстов, переведены на другие языки, если это не оговорено в письменной форме в договоре на поставку программного обеспечения.

Программное обеспечение, описанное в настоящем Руководстве, поставляется по лицензионному соглашению и может использоваться или копироваться только в соответствии с условиями этого соглашения.

Разработчиком и генеральным распространителем программных продуктов Турбо 9 является ЗАО "ДИЦ".

Адрес: 125057, Москва, Чапаевский пер., д. 6, стр. 1

Телефоны для справок: **(499) 157-08-20, (499) 157-04-72, (495) 956-12-50**

Телефоны для консультаций зарегистрированным пользователям:

(499) 157-03-15, (499) 157-03-64

Факс: **(495) 913-2041**

E-Mail: **tb@dic.ru** (для писем), **hotline@dic.ru** (для консультаций)

Web: **<http://www.dic.ru/>**

ЗАО "ДИЦ" 1991-2008

Программа предоставляет квалифицированному пользователю-программисту целый арсенал мощных средств программирования для разработки собственных уникальных прикладных проектов, которые используются пользователями в режиме сессии. Прикладные проекты, созданные с помощью программного средства Студия, по сути, мало чем отличаются от обычных прикладных программ.

Основополагающий принцип работы программы заключается в проектировании и последующем использовании взаимосвязанных проектов различной направленности, главенствующую роль в которых играет финансовый, управленческий и оперативный учет хозяйственной деятельности. Прикладные проекты могут создаваться как самими разработчиками программы, так и сторонними разработчиками.

Основная работа во время проектирования ведется с помощью редактора проекта и заключается в подготовке совокупности настроечных файлов (структура учета, типовые операции, документы, бланки и т.д.). Для разработки проектов в программе имеются встроенный компилятор и [встроенный отладчик](#), а также набор встроенных языков для описания структуры данных (MTL), структуры учета, типовых операций. Язык MTL служит для описания структур данных, с которыми в конечном итоге оперирует прикладной проект. Для написания исходных текстов проекта используется объектно-ориентированный язык ТБ.Скрипт. Все конструкции языка подробно рассмотрены в разделе "[Программирование на языке ТБ.Скрипт](#)". В этом же разделе приводятся сведения для [начинающих программистов](#).

Все сведения, касающиеся разработки прикладных проектов, приведены в подразделах текущего корневого раздела:

- [Встроенный редактор проекта](#)
- [Язык описания модели данных - MTL](#)
- [Объектно-ориентированный язык ТБ.Скрипт](#)
- [Иерархия классов языка ТБ.Скрипт](#)
- [Структура учета;](#)
- [Язык типовых операций](#)
- [Общие приемы разработки проектов](#)
- [Визуальный редактор шаблонов](#)
- [Проектирование бланков](#)
- [Разработка картотек](#)
- [Разработка отчетов на шаблонах](#)
- [Проводки и полупроводки](#)

Прикладные проекты, созданные инструментальными средствами программы в режиме Студии, по сути, мало чем отличаются от обычных прикладных программ. Они точно также проходят этапы разработки, отладки, распространения среди пользователей, штатного функционирования, усовершенствования и далее вновь распространения и функционирования. В некоторых случаях проекты достигают стадии закрытия, если возникает необходимость существенной переработки самой идеологии построения проекта.

Программа, являясь одновременно и средством разработки, и средой исполнения проектов, имеет широкий набор средств для управления развитием проекта на всех этапах его функционирования. В данном разделе рассматриваются программные средства, предназначенные для разработки и модификации проекта как единого целого:

[Создание нового проекта](#)
[Цикл проектирования](#)
[Редактор проекта](#)
[Режимы работы программы](#)
[Настройка параметров обработки](#)

Иными словами, речь в этой главе пойдет о проектировании на макроуровне, без углубления в детали реализации.

Поскольку проект, как правило, состоит из большого числа разнородных сущностей (таких как план счетов, аналитика, журналы, бланки, картотеки и т.д.), каждая из них имеет свои правила описания, редактирования и применения, а потому требует отдельного рассмотрения. Разработке специфических составных частей проекта посвящены отдельные подразделы Справочной системы, в том числе -

- [структура учета,](#)
- [прикладные классы,](#)
- [бланки,](#)
- [картотеки,](#)
- [внутренние отчеты,](#)
- [переменные,](#)
- [аналитические признаки,](#)
- [язык типовых операций,](#)
- [журналы хозяйственных операций.](#)

Проекты - это независимо разработанные подсистемы, отличающиеся своей функциональностью и содержащие индивидуальный набор документов, бланков, картотек, операций и т.д. В проектах сконцентрирована вся информация об объектах, необходимая для автоматизации той или иной сферы деятельности.

Для облегчения работы с разнообразными объектами, входящими в состав проекта, Студия предоставляет удобный интерфейс с общими принципами управления и единообразным отображением всех объектов в окне редактора проекта, которое открывается сразу же после загрузки проекта в режиме проектирования. Если это окно было закрыто, его можно вновь открыть командой [Проект|Редактор проекта](#), горячей клавишей **F11** или инструментальной кнопкой страницы "Проект".

Иерархическая структура объектов различных типов представлена в левой части окна редактора. Причем несколько объектов или даже группы объектов одного типа объектов могут описываться в одном файле, в то время как объекты некоторых других типов могут описываться сразу в нескольких файлах. Кроме того, в Студии существуют и такие объекты, которые полностью хранятся в информационной базе, то есть не имеют представления в виде файлов.

Для получения сведений о работе с проектами предназначены следующие темы:

- [Диалог "Открыть проект"](#)
- [Окно редактора проектов](#)
- [Добавление группы объектов](#)
- [Команды редактирования элементов проекта](#)
- [Свойства проекта](#)
- [Подпроекты. Надпроекты. Лицевой проект](#)
- [Создание объектов проекта](#)
- [Создание бланка](#)
- [Создание картотек](#)
- [Создание нового класса](#)
- [Создание отчета](#)
- [Редактирование отчета](#)
- [Схемы доступа](#)
- [Просмотр записей](#)
- [Структура учета](#)

Данный диалог позволяет выбрать проект для загрузки его в редактор проекта. В стандартной настройке интерфейса программы диалог открывается по команде меню [Файл|Открыть проект](#).

В диалоге пользователь должен выбрать сервер, а затем – конкретный проект из числа тех, что имеются на этом сервере. Если нужного сервера еще нет в списке, его можно добавить.

В левой части диалога "Открыть проект" расположен иерархический список, содержащий имена серверов данных и прикладных проектов, установленных на них. По умолчанию, список содержит лишь один сервер – "Мой компьютер". Для того чтобы развернуть ветвь с проектами сервера, необходимо выполнить двойной щелчок мышью, наведя курсор на требуемый сервер, или нажать клавишу **серый плюс '+'**. Свернуть ветвь какого-либо сервера можно также с помощью мыши или по нажатию клавиши **серый минус '-'**.

Для регистрации нового сервера в списке необходимо нажать кнопку **Добавить** или выполнить команду **Добавить сервер (Ins)** контекстного меню. В результате выполнения этой команды открывается диалог ["Добавить сервер"](#), в котором пользователь выбирает из выпадающего списка один из доступных компьютеров сетевого окружения. Выбранный компьютер добавляется в список без проверки того, является ли он сервером данных (т.е. на нем должны быть установлены и зарегистрированы серверные модули ядра программы, а также выполняться сервисы, обеспечивающие работу сервера). Если компьютер не является сервером данных, то попытка просмотреть список его информационных баз приведет к ошибке.


Для удаления сервера из списка следует его предварительно выделить, а затем выполнить команду **Удалить сервер (Del)** контекстного меню или нажать кнопку **Удалить**.

Переход в режим проектирования происходит в случае выбора проекта кнопкой **Выбор** или двойным щелчком мыши на каком-либо из проектов.

По нажатию кнопки **Настройка** открывается диалог [с настройками соединения](#) выбранного сервера.

Назначение диалога: автоматизация процесса создания бланков под руководством Мастера, который позволяет быстро и безошибочно создать бланк. Переход между страницами диалога выполняется кнопками **Далее>** или **<Назад**. Процесс разработки бланка в любой момент можно прервать, нажав кнопку **Отмена**.

На первой странице диалога необходимо ввести общие сведения о бланке, заполнив следующие поля ввода (см. рис. Задание общих сведений бланка при его создании):

- **Имя** - имя бланка, которое используется внутри программ на ТБ.Скрипт и выводится в иерархии проекта;
- **Описание** - описание бланка, поясняющее назначение бланка, которое будет отображаться в качестве заголовка окна бланка;
- **Унаследован от** - необязательное поле, задающее [базовый класс](#), от которого новый бланк будет наследовать свойства. Ввод в поле производится с помощью диалога "Формы бланков". Наследование можно задать от любого существующего в проекте класса бланка. Если проект имеет подпроекты, то наследование может осуществляться и от бланков подпроектов. В этом случае имена таких бланков указываются полностью вместе с именем подпроекта, в то время как для бланков текущего проекта его имя опускается.
Класс, заданный в этом поле, указывается в первой строке файла *.cod после слов class inherited, а далее в кавычках записывается текст, введенный в поле **Описание**;
- **Шаблон** - необязательное поле для задания шаблона вручную или из списка (кнопка ) , на основе которого будет создан шаблон бланка. Список формируется на основе TPL-файлов, имеющих в папке Templates\Blanks. При наличии в папке одноименного COD-файла новый бланк получает не только указанный шаблон, но и имеющийся исходный код, который корректируется в соответствии с актуальными именами задействованных классов.

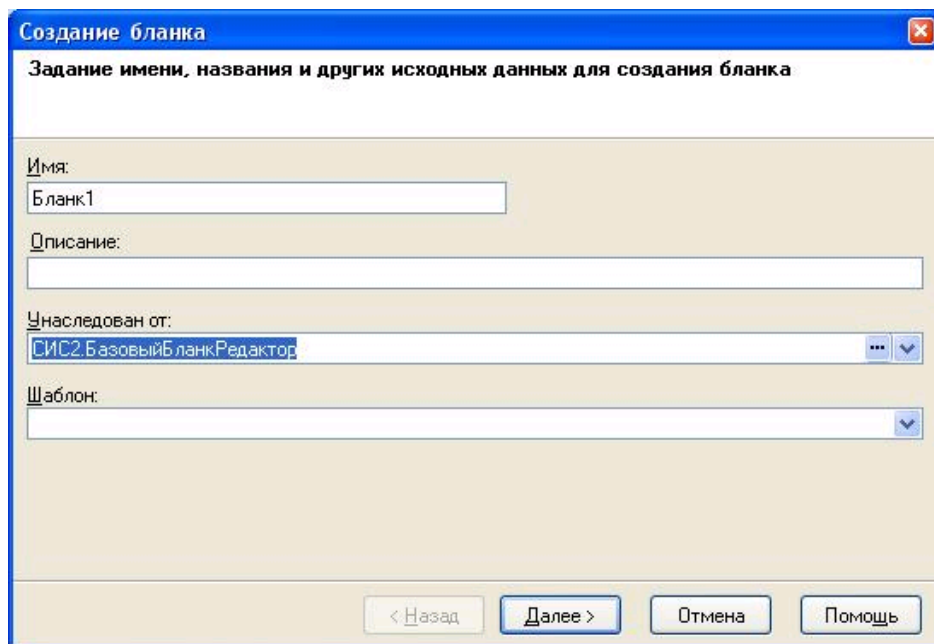



Рис. Задание общих сведений бланка при его создании.

На второй странице следует определить, является ли создаваемый бланк бланком-редактором некоторой записи. Если выбрана радио-кнопка **Бланк не является редактором записи**, то все элементы диалога становятся недоступными и после нажатия кнопки **Далее** открывается последняя страница диалога. Если бланк редактирует запись, то ее необходимо указать в поле **Доступные записи**. Поле заполняется путем выбора записи из иерархического списка диалога "Классы записей", который открывается кнопкой .

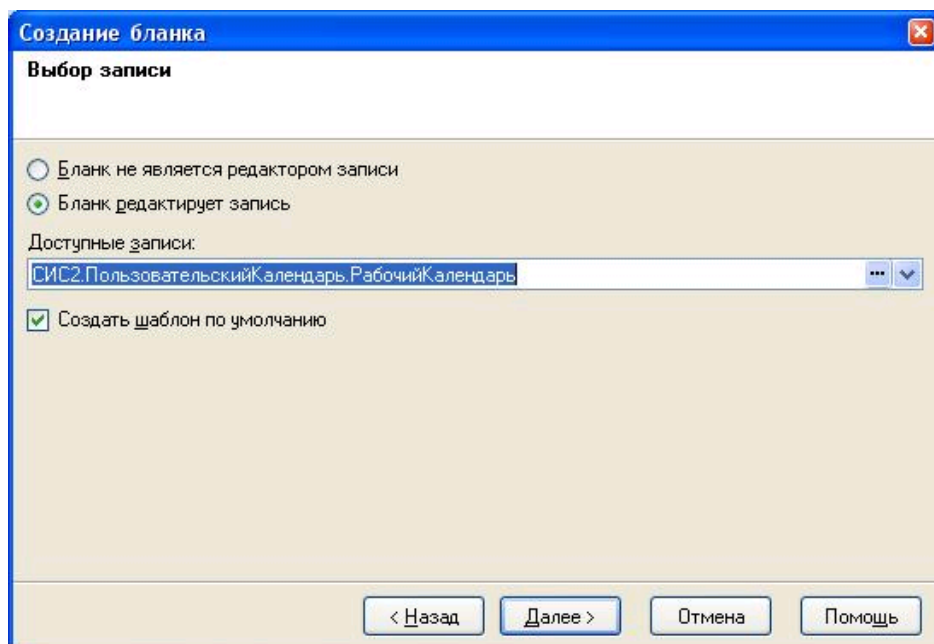


Рис. Выбор записи.

Если включить флаг **Создать шаблон по умолчанию**, на форме бланка (в TPL-файле) будут автоматически созданы поля для ввода значений в поля редактируемой записи. При этом, если бланк формируется на основе некоторого непустого шаблона, то все автоматически добавляемые секции помещаются ниже тех, что есть в шаблоне.

На третьей странице необходимо указать, какие поля записи следует включить в форму бланка. В шаблон включаются только те поля, слева от которых установлен флаг ☒.


Последняя четвертая страница завершает процедуру формирования и регистрации в проекте нового бланка. После нажатия на кнопку **Создать** будет сформировано три файла с расширением *.cod, *.tpl и *.bro с именем, заданным в поле **Имя**. Если был установлен флаг **Открыть редактор бланка**, то непосредственно после создания бланка он открывается на редактирование.

Вызов диалога выполняется командой **Добавить (Ins)** из [окна редактора проектов](#). При этом курсор должен быть установлен на группе *Формы бланков* или на ее объекте.

Назначение диалога: автоматизация процесса создания картотеки под руководством Мастера, который позволяет быстро и безошибочно создать картотеку. Переход между страницами диалога выполняется кнопками **Далее** или **<Назад**. Процесс разработки картотеки в любой момент можно прервать, нажав кнопку **Отмена**.

Внимание. Перед созданием картотек необходимо выполнить [предварительные операции](#).

На первой странице необходимо ввести исходные данные для создания картотеки, заполнив следующие поля ввода:

- **Имя** - обязательное поле, в котором указывается имя картотеки (имя файла). Это имя используется внутри программ на ТБ.Скрипт и выводится в иерархии проекта;
- **Описание** - название картотеки, которое будет отображаться в качестве заголовка окна картотеки;
- **Унаследован от** - необязательное поле для указания [базового класса](#), от которого новая картотека будет наследовать свойства. Ввод в поле производится с помощью диалога "Формы картотек". Наследование можно задать от любого существующего в проекте класса картотеки. Если проект имеет подпроекты, то наследование может осуществляться и от картотек подпроектов - имена таких картотек указываются полностью, вместе с именем подпроекта, в то время как для картотек текущего проекта его имя опускается. Класс, заданный в этом поле, указывается в файле *.cod в первой строке кода после слов class inherited, а далее в кавычках записывается текст, введенный в поле **Описание**;
- **Шаблон** - необязательное поле для задания шаблона, на основе которого будет создан шаблон картотеки. Поле заполняется вручную или из выпадающего списка (кнопка ) , который формируется на основе TPL-файлов, имеющихся в папке Templates\Cards. При наличии в папке одноименного COD-файла класс картотеки формируется не только с учетом указанного шаблона, но и имеющегося исходного кода, который корректируется в соответствии с актуальными именами задействованных классов.

На второй странице необходимо указать запись (файл с MTL-описанием [модели данных](#)), которая отображается в окне картотеки. Поле **Доступные записи** заполняется путем выбора записи из иерархического списка, представленного в диалоге "Классы записей". Если включить флаг **Создать шаблон по умолчанию**, то на форме картотеки в TPL-файле будут автоматически созданы поля для ввода значений в поля редактируемой записи. При этом, если бланк формируется на основе некоторого непустого шаблона, то все автоматически добавляемые секции помещаются ниже тех, что есть в шаблоне.

Третья страница предназначена для определения полей, которые следует включить в форму картотеки. Можно включить все поля, установив радио-кнопку **Все поля записи**. Если включена радио-кнопка **Только выбранные поля**, то в шаблон включаются те поля, для которых установлен флаг. В этом случае кнопка **Выключить все** устанавливает флаги во всех полях, а кнопка **Включить все** снимает все флаги.

Последняя, четвертая страница завершает процедуру формирования и регистрации в проекте новой картотеки. После нажатия на кнопку **Создать** будет сформировано три файла с расширением *.cod, *.tpl и *.bro с именем, заданным в поле **Имя** данного диалога. Если был установлен флаг **Открыть редактор картотеки**, картотека сразу же открывается на редактирование.

Вызов диалога выполняется командой **Добавить (Ins)** из [окна редактора проектов](#). При этом курсор должен быть установлен на группе *Формы картотек* или на объекте из этой группы.

Название диалога зависит от вида добавляемого объекта. Данный диалог предназначен для добавления класса записи, учетной операции, интерфейсной схемы, схемы доступа и шаблонов репликации. Вызов диалога выполняется командой **Добавить (Ins)** из [окна редактора проектов](#). При этом курсор в иерархии объектов должен быть установлен на нужной группе или на объекте из этой группы.

Замечание. Для добавления в проект журналов, справочников, общих переменных курсор должен быть установлен на объекте Структура учета.

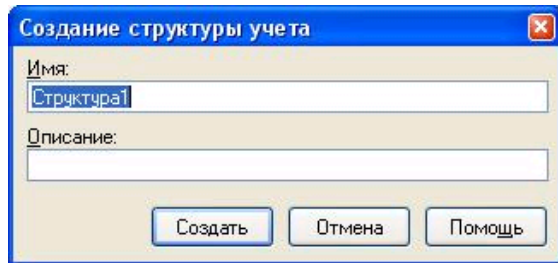


Рис. Создание объектов проекта.

Поле **Имя**

В поле необходимо ввести имя объекта, уникальное в контексте всего проекта. Фактически, это идентификатор, под которым данный объект будет доступен из программ на ТБ.Скрипт. Поле является обязательным для заполнения, так как данное имя используется также в качестве имени файла на диске.

Поле **Описание**

В поле указывается произвольная строка (комментарий), поясняющая назначение вновь создаваемого объекта. Поле можно не заполнять.

Кнопка **Создать**

Для включения нового элемента в проект необходимо нажать кнопку **Создать**. В результате также создается файл с именем, заданным в поле **Имя**, расширение которого зависит от типа объекта. Файл размещается в той папке (группе), на которой был установлен курсор при открытии диалога. Например, если создавались объекты Структуры учета, то на диске создается текстовый файл с расширением *.lis.

Создание шаблона репликации

Если создается [шаблон репликации](#), то после нажатия на кнопку **Создать** диалог закрывается и открывается редактор шаблонов репликации, который позволяет выполнить только настройку репликации записей, в отличие от редактора схем репликаций, который имеет две страницы.

В левой части редактора шаблонов располагается иерархический список всех записей (классов документов) текущего проекта и других лицевых проектов, используемых текущим проектом. Каждую запись, участвующую в репликации, можно настроить отдельно с точностью до поля, выделив ее в иерархии. При этом в правой части окна выводится список всех полей записи, в котором с помощью флагов можно явным образом задать, какие поля следует копировать, а какие - нет.

Созданный шаблон можно просмотреть в гипертекстовом формате *.xml и отредактировать, выполнив команду **Как текст** контекстного меню.

Работа с данным редактором подробно описана в теме ["Редактор схемы репликации"](#).

Для открытия диалога "Добавить группу объектов" необходимо в окне редактора установить курсор на имени проекта и нажать клавишу *Ins* или выполнить команду **Добавить** контекстного меню (см. рис. Группа объектов).

Внимание! Команда доступна, если разрешено редактирование текущего окна, т.е. поле **Т/Ч** в [строке состояния](#) не активно. В этом случае заголовок поля не выделен, и отображается серым цветом.

Диалог "Добавить группу объектов"

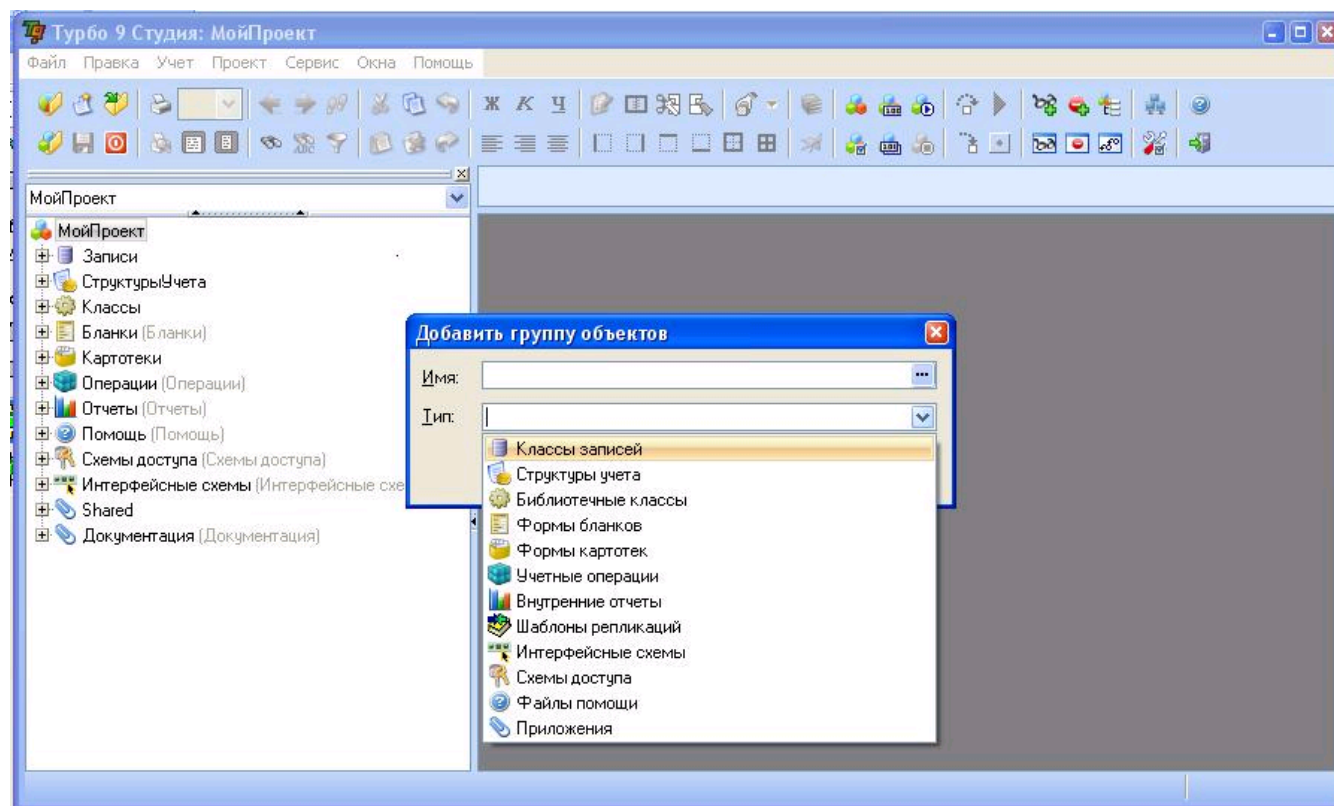


Рис. Группа объектов.

Поле **Имя**

В данном поле указывается уникальное имя группы верхнего уровня, которое должно отражать назначение вводимой группы объектов.

Поле **Тип**

В этом поле указывается тип объекта, заданный в выпадающем списке, который приведен на рис 1. Список открывается кнопкой

Например, для группы объектов бланков следует в поле **Имя** ввести название группы Бланки, в поле **Тип** из выпадающего списка выбрать "Формы бланков".

Кнопка **Добавить**

Для включения группы объектов в проект необходимо нажать кнопку **Добавить**. В результате на диске создается папка, имя которой совпадает с именем, указанным в поле **Имя**. Вновь созданная папка (в нашем случае Бланки) будет размещена в папке Project\МойПроект, где МойПроект - название текущего проекта.

Типы объектов

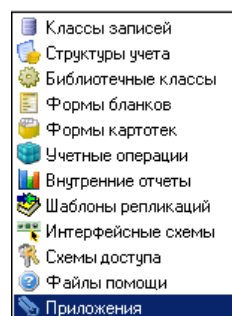



Рис. 1

Каждому типу объектов соответствуют следующие типы файлов:

- **Классы записей** - папка [с группой MTL-файлов](#), описывающих модели данных в формате *.MTL;
- **Структуры учета** - папка с LIS-файлами, содержащими описание [структуры учета](#);
- **Библиотечные классы** - папка с COD-файлами [пользовательских классов](#);
- **Формы бланков** - папка с [бланками](#), описанными в COD- и TPL-файлах;
- **Формы картотек** - папка с [картотеками](#), описанными в COD-, TPL- и BRO-файлах;
- **Учетные операции** - папка с COD-файлами с описаниями [типовых операций](#);
- **Внутренние отчеты** - папка с файлами настроек [встроенных отчетов](#) в формате *.cod;
- **Шаблоны репликации** - файлы с заготовками схем [репликации](#) в формате *.xml;
- **Интерфейсные схемы** - файлы с настройками [интерфейса программы](#) в формате *.shi;
- **Схемы доступа** - папка с файлами, содержащими описание [прав доступа](#) пользователей, в формате *.shm;
- **Файлы помощи** - папка с файлами помощи в формате *;
- **Приложения** - файлы с пользовательскими приложениями в формате *.tbr.

Для каждой из ветвей иерархии объектов, размещенных в [окне редактора проектов](#), а также для ее элементов, существует набор однотипных команд, которые можно вызывать из контекстного меню или с помощью соответствующих горячих клавиш.

В том случае, когда курсор размещается в левой части окна на каком-либо объекте из иерархии объектов, контекстное меню редактора проекта содержит перечисленный ниже набор команд:

- **Открыть (Enter)** - в зависимости от вида объекта открывает файл с описанием этого объекта, редактор шаблона репликации, диалог настройки отчета, диалог настройки интерфейса и т. д.
- **Открыть с помощью** - открывает диалог "Open with" операционной системы Windows для выбора редактора с помощью которого можно просмотреть файлы помощи.
- **Просмотр записей** - команда доступна, если курсор установлен на группе с MTL файлами  и открывает окно ["Просмотр записей"](#);
- **Просмотр структуры учета** - команда доступна, когда курсор в иерархии объектов установлен на объекте Структура учета и открывает окно для просмотра [свойств объектов структуры учета](#);
- **Добавить (Ins)** - открывает диалог для добавления нового объекта заданного типа. При добавлении журнала, переменной или группы признаков пользователю необходимо в открывшемся [диалоге](#) ввести название объекта, и система сама выполнит все необходимые действия по формированию внутренних структур данных и регистрации их в проекте. В частности, возможно автоматическое создание файлов на диске.

При [добавлении отчетов и классов](#) кроме названия пользователь должен указать каталог, в котором следует создать объект. В случае выполнения команды **Добавить** для бланков и картотек запускается соответственно Мастер [бланков](#) и [картотек](#), со который направляет пользователя в ходе выполнения этой процедуры.

- **Добавить папку (Alt+Ins)** - создает новую папку с именем, заданным пользователем. В результате ее выполнения Студия создает пустой подкаталог в той папке, которую пользователь выбрал из списка имеющихся.

Например, пусть к проекту подключены папки "Бланки" и "BLN", где должны размещаться бланки (это было сделано ранее в окне редактора проекта с помощью команд в иерархии файлов). Если выделить в иерархии объектов ветвь "Бланки" и выполнить команду **Добавить папку**, Студия выведет диалог, где нужно будет ввести название папки, а также выбрать из списка каталог верхнего уровня - в нашем случае это будут "Бланки" и "BLN". После того как имя и размещение определены пользователем, программа создает пустую папку и отображает ее в иерархии объектов проекта. Введение папок позволяет группировать объекты одного типа по смысловому признаку и формировать более стройные с логической точки зрения проекта. В качестве отвлеченного примера можно привести книгу с подробным многоуровневым содержанием. Очевидно, что найти в такой книге нужный фрагмент гораздо проще, чем в равном по объему тексте без единого заголовка.

Благодаря использованию папок объектная модель приобретает ярко выраженную структурированность. Так, если в папке "BLN" из предыдущего примера создается бланк "Накладная", то полностью квалифицированное имя данного бланка будет выглядеть как "BLN.Накладная". Рекомендуется давать папкам, впрочем как и самим бланкам, интуитивно понятные имена. В этом случае текст программ, да и сама иерархия проекта, будут более "читабельными".

- **Переименовать (Ctrl+Enter)** - изменяет названия конкретных элементов иерархии (файлов или папок).
- **Удалить (Del)** - удаляет выделенный объект. Следует иметь в виду, что перед выполнением команды **Удалить** удаляемые объекты и их файлы должны быть закрыты. Если какой-либо объект находится в состоянии редактирования, то удалить его нельзя и выдается сообщение об ошибке.
- **Исключен (Ctrl+-)** - действует по принципу флага, переключаясь между двумя состояниями, и используется для группы объектов. Выполнив ее один раз, пользователь временно исключает группу из проекта и папки не обрабатываются Студией при компиляции и исполнении проекта. При этом на значке группы объектов появляется желтый кружок. Повторное выполнение команды над той же группой вновь включает ее в проект.
- **Поиск (Ctrl+S)** - осуществляет контекстный поиск строки в иерархии файлов или объектов проекта, в результате выполнения которой на экран выводится стандартный [диалог "Поиск"](#) для задания критерий поиска.

Команда **Повтор (Ctrl+L)** предназначена для продолжения поиска с прежними критериями, начиная с текущего места в иерархии.

Следует отметить, что в иерархии проектов можно найти элемент по его названию, используя так называемый "быстрый поиск" - то есть просто набирая последовательно символы, входящие в искомое имя. Например, нажав клавишу 'и', пользователь инициирует поиск такого элемента, в названии которого имеется (не обязательно в начале слова, но и в середине) буква 'и'. Если хотя бы один такой элемент есть, то выделяется первый из них. Если пользователь затем нажмет клавишу 'н', то шаблоном для

поиска станет уже строка 'ин', при этом выделяется первый из элементов, в название которого содержится подстрока с двумя символами 'ин'. Набор символов можно продолжать до тех пор, пока требуемый элемент не будет найден или пока не появится сообщение "Строка не найдена". Текущая последовательность символов, введенная пользователем для быстрого поиска, отображается в строке состояния.

- **Поиск в файлах** - открывает диалог [Поиск в файлах](#) для контекстного поиска и замены в группе заданных файлов.
- **К файлам** - подключение к программе Проводник (Explorer), причем курсор будет находиться в левой части окна Проводника на папке, которая была выделена перед вызовом команды или в которой размещается выделенный файл.

В программе предусмотрен набор средств для кодирования файлов и защиты программных продуктов от несанкционированного доступа к ним. Для этих целей каждому программному продукту присваивается индивидуальный номер лицензии. Номера лицензий на программные продукты (Ядро, Студия и прикладные проекты) запрашиваются при установке программы. Добавить номер лицензии можно также при администрировании системы.

Работать с программой в [режимах](#) сессии и разработки, за исключением демо-режима, пользователь может только при наличии лицензии на инструментальное Ядро программы и электронного ключа, указанного в лицензии. Для этого пользователь перед запуском программы должен подсоединить входящий в комплект поставки электронный ключ к порту компьютера. Дополнительно для работы в режиме сессии требуется лицензия на каждый прикладной проект (входящий в состав информационной базы), с которым будет работать пользователь, а для работы в режиме разработки проектов необходима лицензия на Студию. Следовательно, режим работы программы зависит от кода приобретенной пользователем лицензии.

Все поставляемые программой прикладные проекты кодируются для защиты их от несанкционированного доступа и используются только при наличии лицензий, т.к. при открытии сессии выполняется проверка на наличие электронного ключа и номеров лицензий.

Шаблоны (файлы *.tpl) кодируются по номеру специального ключа, коду проекта и упакованному номеру выпуска, который включает номер года, квартала и обновления проекта. Надпроектам, в состав которых входит проект, код которого используется для кодировки шаблонов, присваивается нулевой код, чтобы не выдавать для них дополнительных лицензий. Использовать шаблоны, закодированные таким способом можно только в составе проекта, код которого совпадает с кодом проекта, которым они были закодированы, в противном случае в сессии они открываться не будут.

Лицензии на программные продукты

При покупке программных продуктов пользователю выдаются на них лицензии и электронный ключ. Номер лицензии состоит из номера ключа, номера продукта и номера выпуска. Каждый программный продукт в стандартной поставке имеет свой уникальный номер. Например, Ядру программы присвоен номер 900, а Студии - 910.

Каждое обновление бланков имеет свой индивидуальный *номер выпуска*, чтобы отразить изменения в бланках первичных документов, происходящих в связи с изменением законодательства или по другим причинам. Номер выпуска программного продукта формируется в следующем формате: ГГКNN или ГКNN, где

- **ГГ** - двузначный номер года, используется для всех лет, начиная с 2010 года, например, для 2010 года в номере выпуска будет указано число 10, для 2012 - 12;
- **Г** - однозначный номер года указывается для 2007, 2008 и 2009 года, т.е. вместо двух последних цифр в номере года указывается только последняя значащая цифра, например, 7, 8 или 9, а цифра ноль опускается;
- **К** - номер квартала, целое число от 1 до 4;
- **NN** - номер обновления программного продукта для заданного квартала, может принимать значения в диапазоне чисел от 01 до 99, т.е. теоретически продукты в течение одного квартала могут обновляться от 0 до 99 раз.

Начальный номер выпуска устанавливается *15 числа последнего месяца каждого квартала года*. Например, 15 июня 2007 года устанавливаем номер выпуска 7201, а 15 сентября 2007 года номер выпуска меняется на 7301. Во всех промежуточные выпусках номер выпуска меняется путем увеличения начального числа на единицу. Именно этот номер и будет отображаться в свойствах бланка. А вот в лицензии будет указан не он, а *максимальный номер выпуска* в текущем квартале, например, 7299. Просмотреть максимальный номер выпуска можно в списке лицензий в поле **Проект** (он отображается после кода проекта и точки) [на странице "Лицензии"](#) в окне сервера.

При проверке на допустимость кода лицензии необходимо, чтобы лицензия удовлетворяла следующим требованиям:

1. Код проекта, заданный в лицензии должен совпадать с кодом проекта, с которым работает пользователь;
2. Указанный в лицензии номер выпуска должен быть равен или меньше номера максимального выпуска.

Если указанные требования не соблюдаются, программа выдает сообщение об ошибке, и сессия не открывается.

Способы лицензирования Студии

Способ лицензирования Студии устанавливается в настройках проектировщика на [странице "Лицензия"](#). По умолчанию, лицензия на Студию, ищется на локальном компьютере среди списка всех [локальных лицензий](#).

Если лицензия на проект в этом списке не найдена, то ее можно ввести в список лицензий, нажав кнопку **Добавить**, т.к. при отсутствии лицензии сессия для этого проекта перестанет открываться или откроется в демо-режиме.

В демонстрационном режиме допускается работа со Студией без ключа. Но, в этом случае, откомпилированные проекты, созданные в такой Студии, могут работать в сессии тоже только в демо-режиме. Напоминаем, что демо-режим возможен только для однопользовательских версий программы, когда при открытии сессии лицензия на Ядро не требуется, если проекты не требуют лицензирования (имеют код = 0).

В локальном режиме при открытии сессии требуется одна лицензия на Ядро программы, а также лицензии на проекты, разработанные с помощью Студии и входящие в состав информационной базы, код проекта которых отличен от 0. Код проекта указывается в [свойствах проекта](#) в поле **Код проекта**.

В многопользовательском режиме, чтобы не ставить ключ на каждое рабочее место программиста, разрабатывающего прикладные проекты, разрешается использовать сетевой сервер лицензий. Для этого в настройках проектировщика на [странице "Лицензия"](#) нужно указать адрес компьютера распределяющего лицензии. На этом компьютере должна быть запущена служба Т9.Сеть, введена многопользовательская лицензия на Студию и установлен ключ.

Окно [редактора проекта](#) открывается сразу же после загрузки проекта в режиме проектирования. Если окно в текущий момент закрыто, то его можно открыть командой [Проект|Редактор проекта](#) (**F11**). Повторное выполнение команды закрывает окно редактора.

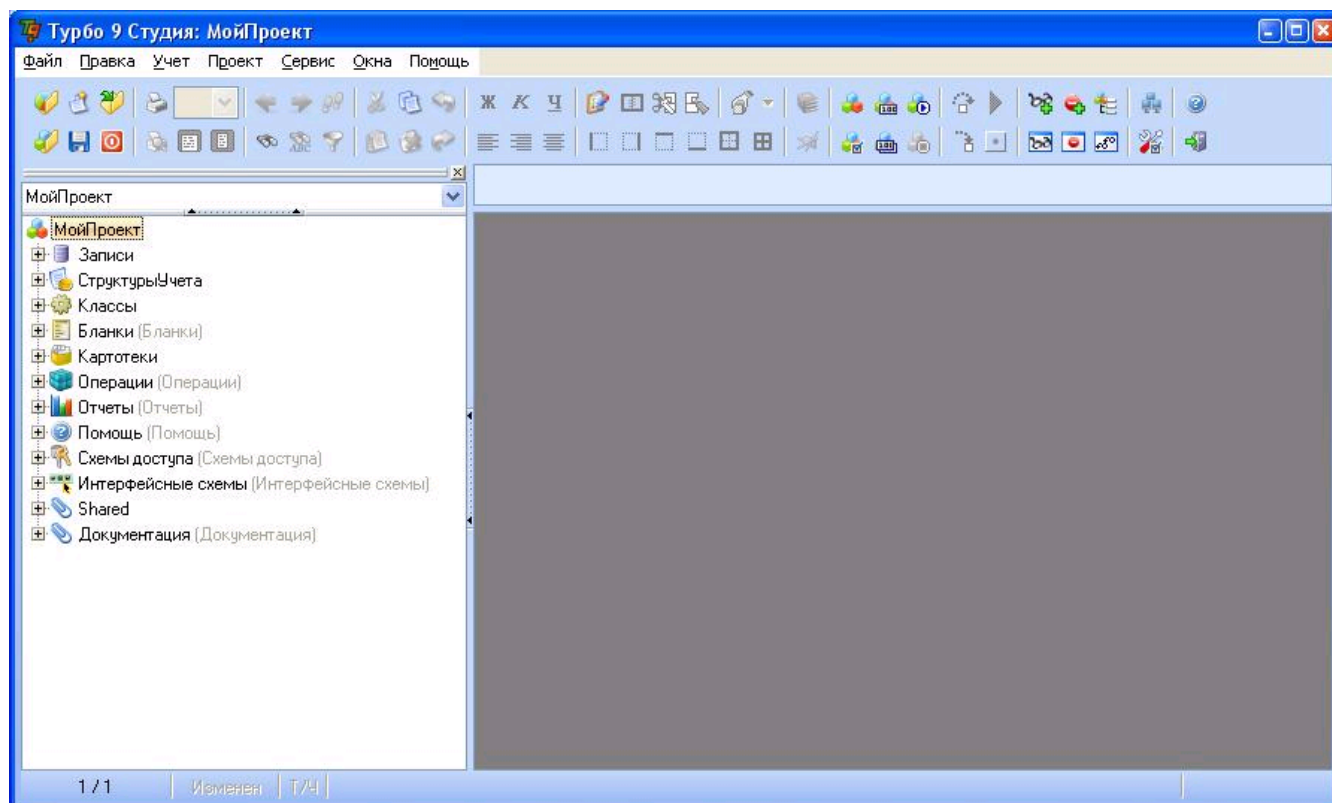


Рис. Окно редактора проекта.

Отображение поля с именем проекта

В левой части окна редактора представлена иерархическая структура [всех объектов](#) различных типов текущего проекта. По умолчанию при открытии проекта в самом верхнем поле, расположенном над иерархией объектов проекта, отображается имя текущего проекта. Если дважды щелкнуть на имени проекта, то в правой части окна выводятся [все свойства проекта](#).

Если поле с именем проекта не отображается, то для его показа нужно подвести курсор к середине серой перемычки (при этом ее средняя часть окрашивается в желтый цвет, как показано на рисунке)

и щелкнуть мышью на области видимости желтого цвета. Повторный щелчок мышью делает поле невидимым.

Переход к подпроекту

Для перехода к [подпроекту](#), если он входит в состав текущего проекта, следует выбрать его имя из выпадающего списка (кнопка ▾). В результате имя выбранного подпроекта отобразится в верхнем поле редактора проектов, а в иерархии объектов показываются объекты выбранного подпроекта, что обеспечивает доступ к ним. Такой способ позволяет легко переходить от одного подпроекта к другому для его просмотра и редактирования.

Иерархия объектов проекта

В иерархию объектов проекта могут входить следующие элементы верхнего уровня:

- [классы записей](#) - иерархия классов документов, построенная в соответствии с MTL-описаниями;
- [структуры учета](#) - перечень LIS-файлов с описанием аналитических справочников, планов счетов (и входящих в них счетов), журналов хозяйственных операций и общих переменных, зарегистрированных в проекте;
- [библиотечные классы](#) - иерархия пользовательских классов, описанных в COD-файлах;
- [формы бланков](#) - иерархия классов бланков, описанных в файлах *.COD и *.TPL;
- [формы картотек](#) - иерархия классов картотек, описанных в файлах *.COD, *.TPL и *.BRO;
- [учетные операции](#) - классы типовых операций, сформированные на основе описаний из файлов *.COD;
- [внутренние отчеты](#) - перечень встроенных отчетов, настроенных для данного проекта;
- [шаблоны репликации](#) - XML-файлы с заготовками схем репликации, используемыми при эксплуатации распределенных проектов;
- [интерфейсные схемы](#) - настройки интерфейса программы (специфичные для проекта) в файлах *.SHI;
- [схемы доступа](#) - описание прав доступа пользователей и их групп в файлах *.SHM;
- [файлы помощи](#) - перечень файлов помощи с расширением *.htm, относящихся к текущему проекту;
- [приложения](#) - перечень файлов приложения, относящихся к проекту.

Объект любого из вышеперечисленных типов может быть создан, отредактирован или удален с помощью [соответствующих команд](#) контекстного меню, вызываемого правой кнопкой мыши.

Следует отметить, что однотипные объекты (например, формы бланков) в иерархии могут быть представлены в виде групп/подгрупп, которые фактически повторяют иерархию папок на диске, причем имена групп совпадают с именем папок, а имена объектов этих групп совпадают с именами файлов. Несколько объектов из одной группы объектов одного типа могут описываться в одном файле (например, записей или объектов структуры учета), в то время как объекты некоторых других типов могут описываться сразу в нескольких файлах.

Весьма удобной возможностью при работе в окне редактора проектов является поддержка технологии Drag'n'Drop ("перетаски и отпусти"). Пользователь может "перемещать" однотипные объекты из одной папки в другую или менять порядок следования объектов в папке. Если при этом держать нажатой клавишу **Ctrl**, то будет создан дубликат объекта с новым именем, причем с сохранением основных свойств образца.

Проекты в Студии разрабатываются независимо друг от друга, такая технология значительно упрощает процесс их разработки. В дальнейшем в случае необходимости разрешается включать заранее разработанный проект в состав другого более сложного проекта, что позволяет создавать проекты любого уровня сложности.

На практике довольно часто встречается ситуация, когда часть требуемой бизнес логики и моделей данных уже описана в каком-либо проекте. Можно было бы скопировать исходные файлы этого проекта в новый проект, однако Студия предлагает более простой и технологичный вариант - использовать готовый проект с нужным функционалом в качестве составной части нового проекта. Это также позволяет конструировать большие проекты на базе нескольких подпроектов меньшего масштаба. Таким образом, серьезный проект Студии можно представить как набор *подпроектов*, причем каждый из них, скорее всего, немного подправлен, усовершенствован и заключен в своего рода оболочку, которая не позволяет снаружи видеть внутреннее устройство всего сооружения.

Итак, *подпроект* - это проект, являющийся составной частью другого проекта. Причем, в проекте можно использовать *все свойства и методы подпроектов*.

Некоторые подпроекты могут изначально разрабатываться не как самостоятельные прикладные системы, а только для использования их в составе других, более крупных проектов. Например, имеет смысл вынести в такой подпроект все, связанное с расчетом налогов, так как данный участок учета будет востребован во многих системах (торговля, зарплата, страхование). Подпроект может быть даже не приспособлен для интерактивного взаимодействия с пользователем. К таким проектам относится проект СИС2, который входит в состав большинства проектов Студии, например, проект Персонал. Подпроекты, используемые проектом, указываются в [свойствах](#) этого проекта.

В ряде случаев при рассмотрении связки проект - подпроект удобнее употреблять другой термин - *надпроект*, под которым понимается проект по отношению к проектам, входящим в его состав. Так, надпроектом является проект Персонал по отношению к проекту СИС2.

Под *лицевым* понимается самостоятельный проект, который может использоваться автономно и приспособлен для интерактивного взаимодействия с пользователем. Лицевой проект может также входить в состав более сложного проекта.

Назначение окна: просмотр всех классов записей. Функциональность и внешний вид окна зависит от режима работы программы. В режиме разработки проектов в окне отображается структура всех классов записей, описанная в текущем проекте и во всех входящих в него [подпроектах](#), а при [вызове окна](#) из сессии - структура, существующая в информационной базе.

На левой панели окна показывается иерархия всех классов записей. В корневой группе указываются имена прикладных проектов и встроенного проекта с предопределенным именем **Kernel**. Правая панель состоит из двух страниц "[Данные](#)" и "[Структура](#)", переключение между которыми осуществляется с помощью закладок в нижней части окна. Внешний вид правой панели зависит от выделенной на левой панели записи или группы записей, а также и от выбранной страницы.

Страница "Данные"

На странице "Данные" отображается содержание выделенной записи, страница доступна только *в режиме исполнения проекта*.

Встроенный проект Kernel

[Установка прав](#) пользователя и их [привязка к пользователям](#) производится путем заполнения полей записи картотеки **Kernel.Settings.Role** и **Kernel.Settings.User**.

Для подключения пользователя к информационной базе (ИБ) его следует зарегистрировать в иерархической картотеке **Kernel.Settings.User**. На каждого пользователя отводится отдельная запись. Ввод нового пользователя выполняется командой **Добавить (Alt+Ins)**, которая доступна при нахождении курсора в правой части окна, где перечислены пользователи.

Назначение полей и флагов картотеки **Kernel.Settings.User** для текущего пользователя:

- **Name, FullName** - краткое и полное имя пользователя. Если пользователь подключается к ИБ [через домен](#), его имя записывается в формате: ИмяПользователя@Домен;
- **FullAccess** - при установке флага пользователь обладает правом полного доступа без каких-либо ограничений. Пользователь "Администратор" всегда имеет полный доступ, вне зависимости от состояния этого флага. Если пользователю разрешен полный доступ, то при открытии сессии в диалоге "[Права пользователя](#)" в список доступных ролей автоматически добавляются 2 служебные роли:
\$FullAccess (Полный доступ)
\$SafeMode (Безопасный режим)
- **Disabled** - при снятом флаге все роли пользователя запрещены, т.е. они игнорируются.
- **SID** - строковое поле, для отображения уникального идентификатора (SID - Security Identifier). Поле заполняется в том случае, когда идентификация пользователя при авторизации через домен происходит не по его имени, а по уникальному идентификатору SID. Поле целесообразно заполнять с помощью функции [ChooseUsers](#).

Замечание. При наличии ссылки на отсутствующую запись, не помеченную как удаленная, вместо "nil" выводится полный уникальный номер записи - DocID. Если попытаться уточнить такую ссылку, выдается сообщение об ошибке "Запись находится в некорректном состоянии...".

В картотеке **Kernel.Отчеты** отображается информация по общим отчётам. Пользовательские шаблоны общих отчётов хранятся в таблице **Kernel.Settings.CommonFile** в группе Root\Reports, их имена, используемые в файловой системе, записываются в поле **Name**. Причем, редактировать картотеку можно на любом компьютере, использующем эту информационную базу (ИБ), даже, если физические файлы размещаются на другом компьютере. Все сделанные изменения сохраняются в файле *.tpl с пользовательскими шаблонами, независимо от того, на каком компьютере выполнялись правки.

Физически на диске указанные файлы *.tpl с пользовательскими шаблонами хранятся в папке Общие файлы ИБ хранятся на диске в папке **Common**, путь к которой можно определить с помощью свойства

[КаталогОбщихФайловИБ](#).

Прикладные проекты

Для каждого табличного журнала, описанного в файле *.lis в [структуре учета](#), в классе **TabJur** прикладного проекта заводится отдельная запись, имя которой совпадает с именем журнала, указанным после ключевого слова **Journal|Журнал**. Для хранения дополнительной аналитики в записи имеется подтаблица **Признаки** со следующими полями:

Название - строковое, имя параметра счета;
Значение2 - ссылочное значение;
Расщепление - целое, расщепление по дебету/ кредиту.

Подтаблица **Признаки** используется как источник данных для параметров типовых операций, имеющих тип данных [ХранилищеПараметров](#).

Страница "Структура"

На странице "Структура" при выделении в левой панели группы записей в верхней строке правой панели выводится ее название, совпадающее с именем файла *.mtl, в котором описаны все записи данной группы, и комментарий, а в остальных строках - перечень всех записей группы. Просмотреть структуру нужной записи можно, выделив ее в дереве слева или дважды щелкнув на ней в перечне записей правой панели. Внешний вид структуры записи и набор команд контекстного меню зависит от выбранного значения поля **Тип метаданных**, расположенного в нижней части окна. При выборе значения **Поля, Индексы и Ограничения** в левой панели для выбранной записи отображаются:

- **Поля** - список всех полей выбранной записи, в том числе и [служебных полей](#), если включен флаг слева от названия команды **Служебные поля** контекстного меню. Кроме этого в списке также показываются все [наследуемые поля предков записи](#) при установке флага в команде **Унаследованные поля**. Все отображаемые поля, доступны только для просмотра;
- **Индексы** - список [индексов](#), которые в отличие от полей и ограничений разрешается редактировать в режиме сессии, используя кнопки **Добавить**, **Изменить** и **Удалить** или одноименные команды контекстного меню. Кнопки **Добавить**, **Изменить** и **Удалить** позволяют соответственно добавить новый индекс, изменить существующий индекс или его удалить. Первые две кнопки открывают соответственно диалог ["Добавление нового индекса"](#) или "Изменить индекс", состав и назначение полей диалогов идентичны.

Команды контекстного меню **Служебные индексы, Индексы, добавленные вручную и Индексы, описанные в MTL** позволяют отобразить на панели соответствующие индексы, если слева от этих команд установлен флаг. При установке трех флагов отображаются все индексы, созданные для текущей записи. Если индексы добавлялись вручную, то становятся доступными команды **Пересоздать** и **Пересоздать все**. Эти команды позволяют пересоздать соответственно текущий индекс или все добавленные вручную индексы.

- **Ограничения** - список системных и/или пользовательских [ограничений](#), который зависит от комбинации установленных флагов слева от названия команд **Системные ограничения, Ограничения, описанные в MTL и Ограничения, добавленные программно**.

В режиме проектирования разрешается открывать файл с MTL-описанием записи для его редактирования с помощью команды всплывающего меню **Описание (F4)**.

Вызов окна выполняется:

- в режиме проектирования из окна редактора проекта командой контекстного меню **Просмотр записей**, которая доступна, если курсор установлен на группе с MTL файлами;
- в режиме исполнения проекта (во время сессии) командой [Сервис|Просмотр записей](#).

Отчеты на стадии разработки проекта создаются с помощью [Мастера создания отчета](#), после создания отчета их имена отображаются в иерархии объектов редактора проекта. Такие отчеты называются *проектными*, в отличие от *пользовательских отчетов*, создаваемых самими пользователями во время работы сессии. Созданные проектные отчеты необходимо настроить и отредактировать, установив требуемые параметры отчета. Для этого дважды щелкните на имени нужного отчета, расположенного в левой части [окна редактора проекта](#) в иерархии объектов. В результате в правой части окна редактора проекта появляется форма с элементами управления (полями ввода, флагами, переключателями), аналогичная диалогу ["Внутренние отчеты"](#) в сессии.

Редактирование начинается по нажатию кнопки **Редактировать**, расположенной в нижнем левом углу диалога настройки свойств отчета в редакторе проектов. В результате открывается cod-файл, предназначенный для написания алгоритмической части отчета (дополняющей базовый функционал, "зашитый" в систему). Разработчик имеет возможность отредактировать шаблон и графические настройки отчета, если его формат на [странице "Формат"](#) диалога настройки свойств был выбран соответственно как **Шаблон** или **График**. Переход к tpl-файлу с шаблоном отчета выполняется клавишей **F4**. Шаблон используется как в отчете на шаблоне, так и в [графическом отчете](#). В случае, если отчет строится в формате графика, в окне с шаблоном также отображается область с графиком.

Закончив редактирование нажмите кнопку **Применить**, чтобы записать сделанные установки на диск в файл с расширением *.rpt.

Структура шаблона для отчета в формате шаблона рассматриваются в темах:

- [Разработка шаблона для отчета](#)
- [Шаблон для отчета по оборотам](#)
- [Шаблон для отчета по проводкам](#)
- [Формирование клеток секций отчета на шаблоне](#)

Настройка свойств графического отчета включает два этапа:

[настройку графического отчета](#);
[редактирование надписей в графическом отчете](#).


Следует иметь в виду, что на стадии выполнения проекта (в режиме сессии) пользователи могут [создавать](#) и настраивать собственные, так называемые, пользовательские отчеты. По желанию пользователя часть из них может храниться на локальном компьютере, а часть - в информационной базе. Отчеты, хранящиеся в информационной базе, называются [общими](#) и доступны другим пользователям.

Разработчик проекта также имеет возможность получить доступ к списку общих пользовательских отчетов из кода. Для этих целей следует создать картотеку на базе любого из существующих классов записей, затем удалить из неё все столбцы и в диалоге свойств картотеки ввести в поле **Записи** имя специального системного класса записей: Kernel.Отчеты - именно такой класс записей используется для хранения информации об общих пользовательских отчетах. После компиляции и запуска проекта в этой картотеке станет возможным добавление столбцов для полей [класса записи Kernel.Отчеты](#).

Разработчик имеет возможность сделать любой внутренний отчет доступным для наследования при создании пользовательских отчетов, установив флаг **Разрешено наследовать** на [странице "Формат"](#) диалога "Внутренние отчеты". Подобные наследуемые отчеты перечисляются во время сессии в поле **Базовый** на странице "Формат". Если пользователь выберет из этого списка конкретный отчет, то код его класса и шаблон (но не настройки!) будут унаследованы текущим настраиваемым отчетом. То есть, фактически, пользовательский отчет будет обрабатываться по алгоритмам, написанным разработчиком на ТБ.Скрипт для выбранного базового отчета. Базовый отчет можно выбирать только при настройке пользовательского отчета. Для того чтобы тот или иной внутренний отчет во время сессии появился в списке отчетов, необходимо установить флаг **Виден в дереве отчетов**.

С помощью команды **Удалить** контекстного меню можно физически удалить с диска выделенный проектный отчет из папки или даже саму выделенную папку с отчетами.

По умолчанию имя текущего проекта показывается в самой верхней строке над иерархией объектов в [окне редактора проекта](#). Если оно не отображается, пользователь может самостоятельно [сделать его видимым](#). Если дважды щелкнуть на имени проекта, то в правой части окна выводятся *все свойства проекта*:

- **Название проекта** - уникальное название проекта, характеризующее его назначение;
- **Комментарий** - пояснения к проекту, в частности, если проект создан мастером проектов, то эти сведения автоматически попадут в это поле;
- **Номер версии** - номер текущей версии проекта;
- **Автор проекта** - разработчик проекта и год создания проекта;
- **Код проекта** - уникальный код (номер) проекта, который указывается в [лицензии](#), выдаваемой на программный продукт, и в окне сервера в поле **Проект** на странице "[Лицензии](#)" (для каждой лицензии, перечисленной в окне сервера). Код проекта используется для дополнительной проверки лицензии при открытии сессии и компиляции проектов.
- **Подпроекты** - список всех [подпроектов](#), используемых в текущем проекте. Для ввода нового подпроекта необходимо в выпадающем списке (кнопка ) установить флаг ☒ слева от названия подпроекта и нажать кнопку **Применить**.

Внимание. После ввода нового подпроекта имена всех подпроектов в поле **Подпроекты** автоматически перечисляются в строго определенном порядке, который задает последовательность компиляции.

Все эти свойства проекта могут быть отредактированы прямо в данном окне - достаточно ввести новое значение в соответствующее поле ввода и нажать кнопку **Применить**.

Свойства проекта можно также просмотреть в текстовом файле проекта с расширением *.prj, который открывается командой **Как текст**. Файл проекта создается программой автоматически, но в случае крайней необходимости его можно отредактировать вручную, а затем сохранить изменения, как в обычном файле клавишей **F2**. Все правки вступают в силу только после перезагрузки программы.

Проект, приспособленный для автономного использования (в таком проекте, в частности, должен быть проработан пользовательский интерфейс), называется *лицевым*. Для того чтобы различать лицевые проекты и проекты, которые не могут использоваться автономно, на странице свойств имеется флаг **Лицевой проект**.

Создание и настройка свойств журналов осуществляется в редакторе проекта. Все журналы, используемые в данном прикладном проекте (участке бухгалтерского учета), должны быть перечислены в текстовых файлах с расширением *.lis, которые входят в ветвь Структура учета [иерархии объектов проекта](#). Для редактирования файла, следует дважды щелкнуть на его имени, при этом в правой части окна редактора проекта открывается файл, который можно править, используя текстовый редактор.

Временно удалить ветвь Структура учета из проекта можно командой **Исключен**, при этом журналы не обрабатываются программой, становятся недоступными для пользователей, хотя и остаются в иерархии проекта.

Для добавления журнала необходимо в иерархии объектов проекта выделить ветвь Структура учета, выполнить команду **Добавить** контекстного меню, создать [текстовый файл](#) с расширением *.lis и отредактировать его в соответствии с синтаксисом языка ТБ.Скрипт, указав атрибуты [заголовка журнала](#) и его свойства.

[Описание свойств картотечного журнала](#) содержит указание того, из какой таблицы следует отбирать документы, с каким фильтром, из какого поля брать дату, а также какую типовую операцию для каждого документа вызывать и какие поля документа в какие параметры передавать. Возможно построение журнала по нескольким таблицам (в терминах программы - по нескольким классам записей), вызов для каждого документа нескольких типовых операций, а также - что часто используется - вызов операции не один раз для самого документа, а для каждой его строки. При этом параметрам операции можно присваивать значения, как из шапки самого документа, так и из строки подтаблицы, для которой выполняется операция.

Внимание. После сохранения (клавиша **F2**) изменений, произведенных в файле, и компиляции проекта свойства журналов вступают в силу.

Назначение диалога: создание файла, который содержит код с описанием нового класса.


Поле **Имя**

Поле является обязательным для заполнения. В нем указывается уникальное имя нового класса (идентификатор), являющееся также именем файла. Имя используется для идентификации объекта в иерархии объектов проекта внутри программ на ТБ.Скрипт, а также файла на диске в файловой системе.

Поле **Описание**


В поле указывается произвольная строка (комментарий), поясняющая назначение вновь создаваемого класса. Поле можно не заполнять.

Поле **Унаследован от**

Необязательное поле, позволяет при необходимости выбрать из существующих классов [базовый класс](#), от которого новый класс будет наследовать свойства. Ввод в поле производится с помощью диалога "Библиотечные классы", открываемого кнопкой . Наследование можно задать от любого существующего в проекте или подпроекте класса (если проект имеет подпроекты). В последнем случае имя класса указывается полностью вместе с именем подпроекта, в то время как для классов текущего проекта его имя опускается.

Класс, заданный в этом поле, указывается в первой строке формируемого файла *.cod после слов class inherited, а далее в кавычках записывается текст, введенный в поле **Описание**.

Поле **Шаблон**

В поле указывается шаблон, который позволит автоматизировать процесс написания кода нового создаваемого класса. Во избежание ошибок поле лучше заполнять из выпадающего списка (открывается кнопкой ) , который содержит набор файлов *.cod, имеющихся в папке Templates\Classes.

Кнопка **Создать**

Данная кнопка создает файл с расширением cod и именем, заданным в поле **Имя**. Файл содержит код с описанием нового класса, который в дальнейшем может быть отредактирован в соответствии с [синтаксисом языка](#) ТБ.Скрипт. Файл и размещается в той папке (группе), на которой был установлен курсор при открытии диалога.

Вызов диалога выполняется командой **Добавить (Ins)** из [окна редактора проектов](#). При этом курсор должен быть установлен на группе *Классы* или на объекте из этой группы.


Для создания нового отчета необходимо выделить объект в иерархии объектов проекта в ветви "Внутренние отчеты" [окна редактора проектов](#) и выполнить команду **Добавить** контекстного меню. В результате запускается Мастер создания отчета.

На первом шаге Мастера задаются основополагающие свойства отчета.

В поле **Имя** необходимо ввести уникальное имя отчета. Это имя используется в программах на ТБ.Скрипт как идентификатор класса отчета и выводится в иерархии проекта.

В поле **Название отчета** можно дополнительно указать его расширенное описание, которое будет отображаться в качестве заголовка окна.

Поле **Унаследован от** позволяет, при необходимости, выбрать базовый класс для создаваемого отчета. Наследование можно задать от любого существующего в проекте класса отчета. Если проект имеет подпроекты, то наследование может осуществляться и от отчетов подпроектов - имена таких отчетов указываются полностью вместе с именем подпроекта, в то время как для отчетов текущего проекта его имя опускается.

В поле **Шаблон**, при необходимости, указывается шаблон, который будет использоваться отчетом в формате шаблона (формат отчета выбирается в диалоге настройки свойств отчета и может произвольно меняться как разработчиком, так и пользователем). Шаблон выбирается из списка доступных шаблонов (кнопка ) , который формируется на основе tpl-файлов, имеющихся в подкаталоге Templates\Reports. Если там же расположен и одноименный cod-файл, то новый отчет получает не только указанный шаблон, но и имеющийся исходный код, который корректируется в соответствии с актуальными именами задействованных классов.

Внимание. Если в поле **Шаблон** указано значение *ОтчетПоПроводкам* или *ОтчетПоОборотам*, то второй шаг пропускается.

На втором шаге необходимо выбрать тип создаваемого отчета. *Изменить тип отчета у уже существующего отчета нельзя!* Система предлагает следующие варианты:

- **По проводкам** - отчет строится в табличном виде с разбиением на строки по проводкам или по сводным проводкам;
- **По оборотам** - в отчете показаны остатки и/или обороты, разбитые по параметрам проводки на строки, столбцы или таблицы;
- **По справочнику** - отчет строится в виде разбиения на строки и таблицы по свойствам аналитического справочника;
- **Не определен** - создается отчет, тип которого не определен, в дальнейшем он может использоваться как базовый класс для других отчетов.

Первые три варианта предполагают, что в диалоге настройки отчета, который открывается в правой части окна редактора проекта при выделении в иерархии проекта требуемого отчета, будут установлены прочие настройки. Все эти настройки также могут изменяться и программным образом.

Последний неопределенный тип предназначен только для программной настройки.

На последнем шаге система в указанной папке создает 3 файла с заданным на первом шаге именем и расширениями cod (алгоритмы на ТБ.Скрипт), tpl (шаблон) и rpt (настройки).

Если установить флаг **Открыть редактор отчета**, то можно сразу после создания класса отчета приступить к его [настройке и редактированию](#).

Окно "Структура учета" состоит из нескольких страниц, в каждый момент времени можно просмотреть свойства одного из следующих типов объектов, относящихся к структуре учета:

- [справочники](#)
- [планы счетов](#)
- [типы счетов](#)
- [журналы](#)
- [общие переменные](#)

Переход на нужную страницу для просмотра свойств объекта производится с помощью выпадающего списка, расположенного в левой верхней части окна.

Окно "Структура учета" открывается командой **Просмотр структуры учета**, которая доступна из [окна редактора проектов](#), когда курсор в иерархии объектов установлен на объекте Структура учета.

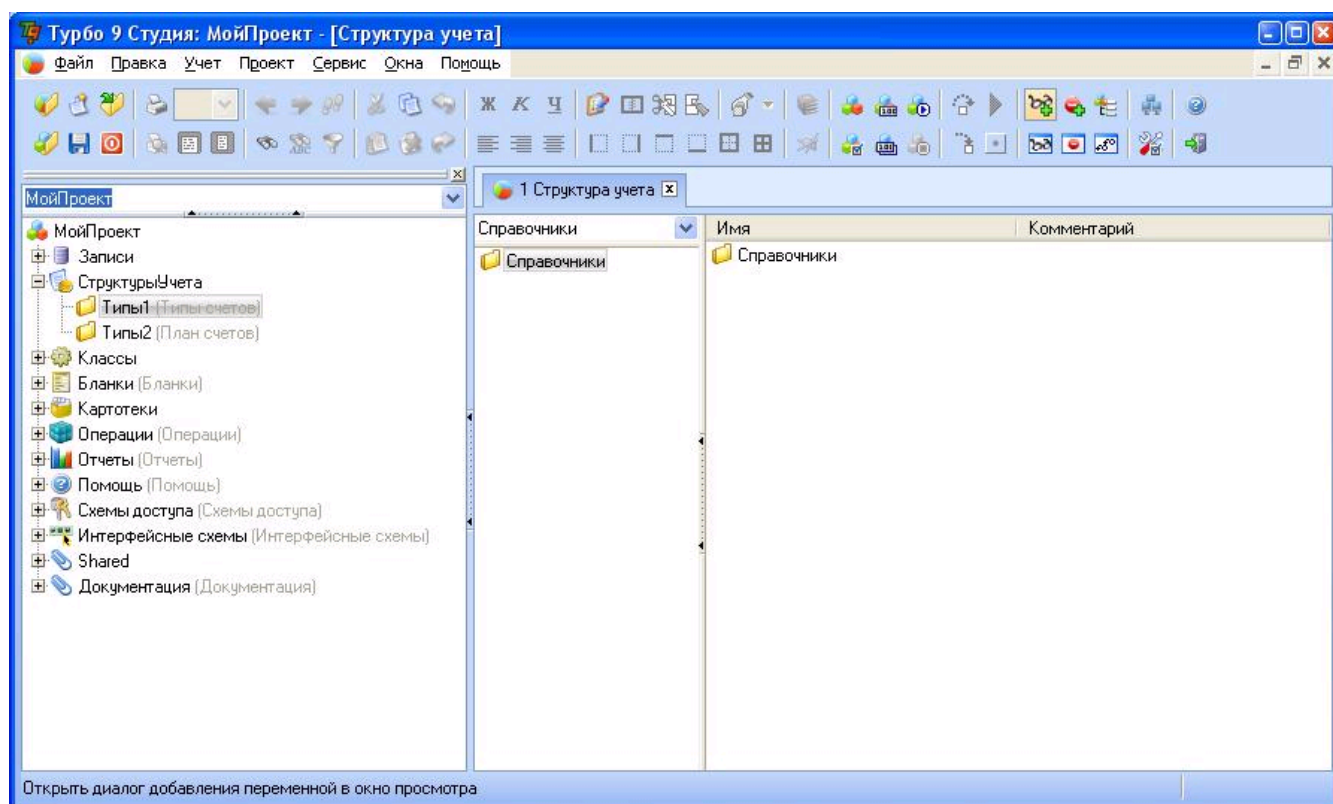


Рис. Окно структуры учета.

Окно состоит из двух частей, в левой части перечислены все объекты данного типа в общем списке или отдельно по каждому подпроекту, входящему в текущий проект (выполнена команда **По проектам** контекстного меню). В правой части окна указываются свойства выделенного объекта или группы объектов.

Непосредственно из текущего окна можно перейти к тому месту в файле структуры учета, где он описан - для этого достаточно его выделить и нажать клавишу **F4** или выполнить команду **Описание** контекстного меню.

Справочники

В левой части окна приведен общий список аналитических справочников или отдельно по проектам. При выделении справочника внешний вид правой части окна зависит от типа метаданных, который можно изменить с помощью выпадающего списка, расположенного в нижней части окна.

Если в качестве типа метаданных выбраны - **Поля**, то правая часть окна содержит три колонки. В первой строке первой колонки **Имя** отображается имя аналитического справочника, в остальных - имена полей справочника, которые указаны в описании [структуры учета](#). В первой строке второй колонки **Тип** указывается назначение справочника (Аналитика, Валюта, Ед. измерения), в остальных строках - тип полей справочника. В третьей колонке выводится комментарий, поясняющий назначение справочника.

Для метаданных - Другое, в правой части окна отображается две колонки, в левой перечисляются названия свойств, в правой - их источники или значения:

- **Документ** - тип документа, класс записи (полный путь к файлу *.mtl в ветви Записи), из полей записи которого берутся значения для свойств элементов справочника.
- **Фильтр** - указывается выражение для фильтра, если он установлен для текущего справочника.
- **Картотека** - картотека документов (источник признаков), во второй колонке указывается полный путь к файлу *.cod

с описанием картотеки в ветви Картотеки.

Типы счетов

На данной странице слева отображается список типов счетов, описанных в проекте, а справа для выделенного типа счета приводится список параметров с указанием их типа и комментария.

Планы счетов

В левой части окна на этой странице выводится список описанных в проекте планов счетов с перечислением всех входящих в них счетов.

Выделение в иерархии любого из планов счетов (счета) приводит к появлению в правой части окна списка счетов данного плана (одного счета), где для каждого счета указывается его идентификатор, тип и комментарий к нему.

Журналы

Список описанных в проекте журналов показывается в левой части окна на странице "Журналы". При выделении журнала в правой части окна перечисляются свойства (поля) журнала с указанием их значений, при выделении группы объектов - приводится список журналов и комментарий к ним.

Общие переменные

На странице "Общие переменные" слева отображается список общих переменных, описанных в проекте, а справа для выделенной переменной указывается ее имя, тип и комментария.

Схемы доступа определяют полномочия пользователей при доступе к общим ресурсам проекта и хранятся в файлах с расширением *.shm. Каждой группе пользователей назначается схема доступа, т.е. одна и та же схема доступа, может использоваться несколькими пользователями одновременно.

Для того чтобы настроить схему доступа для пользователя, в системе должен быть [зарегистрирован](#) хотя бы один пользователь (учетная запись с уникальным именем и паролем), который занесен в [группу пользователей](#). По умолчанию, после установки Студии в ней автоматически создается одна группа "Администраторы" с единственным пользователем "Администратор". Но *схемы доступа по умолчанию не создаются*.

Создание новой схемы доступа

Для создания новой схемы доступа в редактор проекта должна быть добавлена [ветвь "Схемы доступа"](#). Для того чтобы новая схема доступа появилась в иерархии объектов проекта, нужно выделить ветвь "Схемы доступа" и выполнить команду **Добавить** контекстного меню, а в диалоге "Создание схемы доступа" указать уникальное имя (под которым схема будет фигурировать в иерархии объектов) и описание схемы доступа, а затем нажать кнопку **Создать**.

В результате в иерархии объектов в заданной группе создается не только объект, но и - файл (на диске в папке "Схемы доступа" текущего проекта), имя которого совпадает с именем схемы доступа.

Структура файла с описанием схемы доступа

После создания схемы можно приступить к ее редактированию. Двойной щелчок на имени схемы открывает в правой части окна редактора проектов shm-файл, предназначенный для написания директив схемы доступа. Директивы записываются в соответствии со следующим синтаксисом:

```
[common]
title = "Менеджер"
autorun1 = ИмяПроекта.ИмяКласса ИмяМетода
autorun2 = ИмяПроекта.ИмяКласса ИмяМетода
interface = ИмяПроекта.ИмяИнтерфейснойСхемы
HelpContext = ИмяПроекта.ИмяПомощи
```

Директива **title** используется для задания название схемы доступа.

В директивах **autorun1** и **autorun2** указываются следующие данные:

- имена двух процедур автозапуска, первая из них выполняется до считывания настроек Студии, а вторая - после. Может быть указана лишь одна, две или ни одной процедуры;
- имя бланка или картотеки;
- имя Inclass метода, класса. Причем, между именами класса и метода обязательно должен быть пробел.

В директиве **interface** указывается интерфейсная схема, по которой будет происходить вход в систему. Директива **HelpContext** предназначена для указания файла помощи, который вызывается клавишей **F1**.

Внимание! Вновь созданная схема доступа вступает в силу только после [привязки](#) ее к роли, а также после назначения роли конкретному пользователю. Так как пользователи и их роли хранятся [в виде записей](#) в информационной базе, то можно использовать прикладные бланки и картотеки.

Если в проекте не существует специальных прикладных бланков и картотек, то привязка происходит после запуска [сессии](#) (подключения к информационной базе, например, командой [Файл|Открыть сессию](#)).

Создание нового проекта осуществляется из окна администрирования, которое открывается по команде **Сервис|Администрирование (F12)**.

В левой части окна администрирования в виде дерева выводится иерархия объектов программы. В корне иерархии расположен узел "Серверы" и перечислены все компьютеры, зарегистрированные в локальной сети как серверы программы. В простейшем случае список серверов содержит лишь один элемент - "Мой компьютер".

В составе каждого из серверов входят группы "Проекты", "Базы данных", "Информационные базы", "Группы пользователей" и "Пользователи". Все они управляются с помощью команд контекстного меню, вызываемого правой кнопкой мыши, команды которого зависят от выделенной ветви в текущий момент времени.

Всплывающее меню, вызываемое в контексте ветви "Проекты", содержит следующие пункты: **Создать/Добавить проект, Удалить проект, Открыть проект, Опубликовать проект, Установить проект**. По команде **Создать/Добавить проект** открывается Мастер - диалоговое окно, в котором программа шаг за шагом просит пользователя выполнить необходимые действия для создания проекта.

На первом шаге необходимо выбрать одну из опций: создается ли новый проект или добавляется существующий. В первом случае требуется ввести его название и указать каталог, в котором он будет создан. Во втором - программа просит указать имеющийся файл проекта (PRJ-файл).

На втором шаге пользователю предлагается создать по выбору либо чистый проект, либо проект на основе шаблона проекта (шаблон содержит минимальный набор файлов, задающих основу будущего проекта). Также на данном шаге можно ввести справочную информацию о проекте - его название, номер версии, имя автора и произвольный комментарий. Рекомендуется создавать новый проект на основе шаблона (например, "Типовой проект"), так как в этом случае в него переносятся все типовые операции, бланки, отчеты, классы и прочие элементы проекта, имеющиеся в прототипе, что позволяет ускорить разработку и исключает необходимость вручную дублировать часто используемые, полезные наработки.

На последнем, третьем шаге программа спрашивает у пользователя, хочет ли он открыть проект сразу после создания. По нажатию кнопки **Создать** формируется файл с описанием проекта (копируются в указанный каталог файлы проекта шаблона, если он был задан) и происходит его регистрация в системе.

Внимание! В случае, когда данный компьютер используется в качестве сервера данных и на нем в фоновом режиме выполняется модуль в Мастере появляется один дополнительный шаг (между шагами 2 и 3), на котором программа запрашивает сетевой путь к новому проекту в UNC-нотации. Это необходимо для обращения к проекту с других машин. Более подробно вопросы сетевой работы проектов (в частности, о переносе проекта с компьютера на компьютер, удалении проекта и т.д.) рассматриваются в главе [Администрирование](#).

Имеющийся проект можно открывать с помощью команды **Файл|Открыть проект** или путем выбора его из списка последних редактировавшихся проектов, доступного в нижней части меню **Файл**.

После создания нового или открытия существующего проекта клиент переходит в режим проектирования.

Основная работа во время проектирования ведется с помощью визуального редактора проекта и заключается в подготовке совокупности настроечных файлов (структура учета, типовые операции, документы, бланки и т.д.). По завершении очередного этапа осуществляется компиляция: проект проверяется на непротиворечивость, создаются объектные модули, содержащие исполняемый код программ, и двоичные образы прочих настроечных файлов.

При успешном завершении компиляции проект может быть переведен в режим отладки, то есть клиент подключается к информационной базе, использующей данный проект (естественно, она должна быть предварительно создана, см. тему [Администрирование](#)), и запускается на выполнение с возможностью контроля внутренних механизмов работы проекта. В процессе подключения может потребоваться реструктуризация базы (если изменились [MTL-описания](#)).

Итак, стандартный цикл работы клиента при проектировании предполагает следующим действия:

- Переход в режим проектирования командой **Файл|Открыть проект**;
- Редактирование файлов проекта с помощью [редактора проекта](#), [редактора текстов](#), [редактора шаблонов](#), Мастеров (тип инструмента зависит от того, какая часть проекта редактируется);
- Компиляция проекта командой **Проект|Компилировать**;
- Переход в режим отладки командой **Проект|Запустить**;
- Реструктуризация [информационной базы](#) (при необходимости);
- Отладка проекта с помощью [встроенного отладчика](#);
- Возврат в режим проектирования командой **Проект|Остановить**.

По завершении проектирования осуществляется финальная компиляция проекта (без включения [отладочной информации](#)). Результатом компиляции является совокупность двоичных файлов, содержащих в компактном виде информацию об алгоритмах и внешнем представлении данных в проекте. В частности для файла проекта (*.prj) формируется одноименный двоичный файл (*.#prj). В дальнейшем при подключении обычных пользователей используется именно этот файл.

В Студии применена гибкая модель бланков, основанная на раздельном описании их визуальной и алгоритмической частей. Для перечисления переменных бланка, описания его поведения в виде процедур и функций используется язык ТБ.Скрипт, рассмотренный в разделе [Программирование классов на ТБ Скрипт](#).

Внешний вид бланка проектируется с помощью *визуального редактора шаблонов (заготовок бланков)* и сохраняется в отдельном файле, называемом *шаблоном бланка* (см. рис. Визуальный редактор шаблонов бланков).

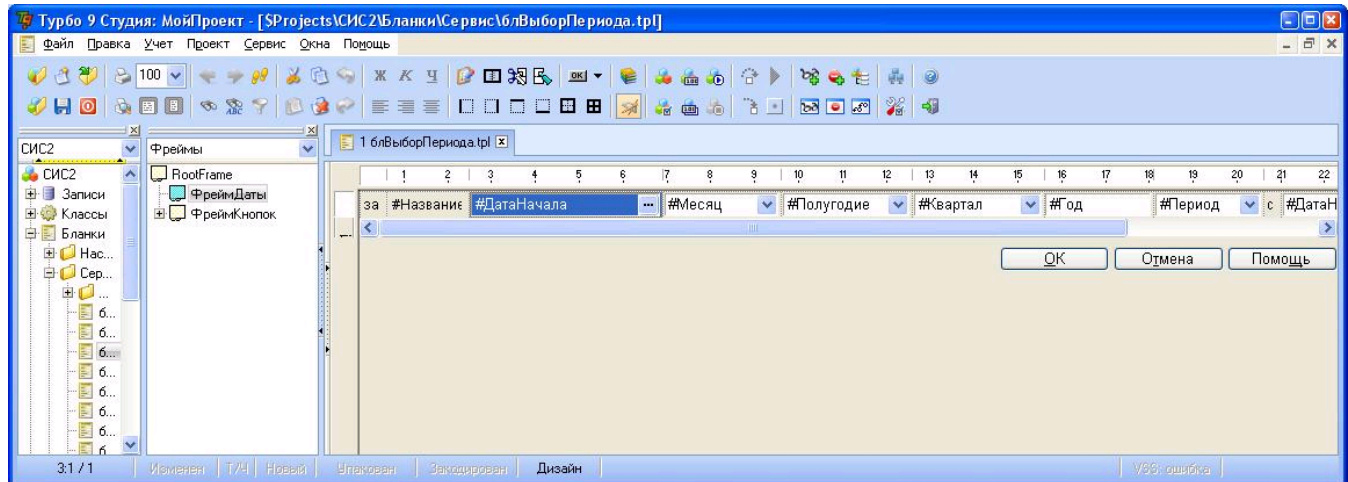


Рис. Визуальный редактор шаблонов бланков.

Такое разделение позволяет использовать при разработке экранных форм бланков все возможности, предоставляемые графической операционной системой Windows, а также упростить и ускорить этот процесс за счет применения специальных инструментальных средств.

Задачи, для которых может быть использован визуальный редактор, не ограничиваются простым дизайном внешнего вида бланка. В процессе визуального редактирования бланку могут быть сообщены свойства, которые будут во многом определять его поведение. Таким образом, визуальный редактор шаблонов является одним из важных средств программирования, используемых в процессе разработки электронных документов проектов Студии.

Следует отметить, что наряду с бланками визуальные формы используются и для представления картотек. Иными словами, не только бланки, но и [картотеки](#), которые рассматриваются в соответствующем разделе, имеют шаблоны. Это обеспечивает унифицированный подход к проектированию интерфейса прикладного приложения и упрощает освоение инструментария, поскольку визуальный редактор применяется как при разработке бланков, так и картотек.

[Режимы работы с шаблоном](#)

[Общие сведения о секциях в шаблонах](#)

[Содержимое клетки секции](#)

[Свойства клетки секции](#)

[Стиль клетки. Библиотека стилей](#)

[Работа с секциями в шаблоне](#)

[События в бланках и их обработка](#)

[Объекты](#)

[Окно настройки свойств](#)

[Окно редактора в дизайн-режиме](#)

[Управление элементами шаблона из процедур и функций](#)

Кроме клеток шаблон бланка может содержать специальные управляющие элементы Windows, называемые объектами. К объектам относятся *кнопка, надпись, флаг, переключатель, редактор, выпадающий список, рисунок, проигрыватель, рамка, OLE-контейнер*. Кроме них доступны также специфические объекты Студии: картотека, дерево картотеки и подтаблица картотеки.

Объект располагается на шаблоне произвольным образом и, как и клетка, может быть связан с переменными бланка. Объекты также имеют собственные наборы событий, на которые можно назначить процедуры-обработчики.

Использование интерфейсных объектов позволяет расширить функциональные возможности бланка, а также сделать работу с ними более удобной и приближенной к стандартам системы Windows.

Более подробно работа с объектами в дизайн-режиме описывается в следующих разделах.

[Добавление объектов](#)

[Выделение объектов](#)

[Удаление объектов](#)

[Принципы функционирования объектов](#)

[Окно настройки свойств](#)

Выделение объектов

Одновременно на шаблоне в дизайн-режиме может быть выделен лишь один интерфейсный объект. Выделить его можно с помощью *клавиатуры* или *мыши*.

В первом случае необходимо последовательно нажимать клавишу *TAB*, перемещая выделение (фокус ввода) с одного элемента шаблона на другой, причем в последовательности элементов учитываются и клетки секций, и объекты.

Для выделения объекта мышью достаточно подвести к нему курсор и выполнить щелчок.

Добавление объектов

Интерфейсные объекты добавляются на шаблон с помощью команд из группы команд **Вставить Объект**, которая в стандартной настройке Студии доступна с панели инструментов (страница "Шаблон").

Каждый тип объекта имеет соответствующую одноименную команду. Например, для того чтобы добавить на шаблон флаг, необходимо выполнить команду **Вставить Объект | Флаг**. После этого требуется подвести курсор мыши к тому месту на шаблоне, где планируется разместить объект, и сделать щелчок мышью.

Объекты, добавленные на шаблон, могут управляться из кода бланка. Для этого необходимо в cod-файле описать переменную с тем же именем, что и добавленный объект, и соответствующего типа. Например, если на шаблон добавлен объект типа **Кнопка** с именем КнЗакреть, то в тексте класса (в разделе **InObject**) необходимо ввести описание переменной:

```
КнЗакреть :Кнопка;
```

Тогда при создании бланка система автоматически проинициализирует эту переменную ссылкой на объект шаблона. Используя свойства и методы встроенного класса **Кнопка** по отношению к данной переменной, можно выполнять различные действия над кнопкой. Например, чтобы изменить текст, выводимый на кнопке, достаточно написать:

```
КнЗакреть.Надпись = "Отмена";
```

Таким образом осуществляется управление объектами из исходного кода.

Существует возможность и обратной связи, то есть управления исполнением кода в зависимости от состояния объектов шаблона. Это реализовано с помощью механизма [обработчиков событий](#).

Для удобства программиста система позволяет автоматически генерировать как описания переменных (для этого достаточно выполнить двойной щелчок мышью в поле **Имя диалога свойств**), связанных с элементами шаблона, так и заготовки обработчиков событий.

Общие принципы использования интерфейсных объектов в режиме исполнения проекта соответствуют стандартным приемам работы с элементами управления Windows.

Удаление объектов

Для удаления объекта его необходимо предварительно [выделить](#) и нажать клавишу *Del* или выполнить команду **Удалить** контекстного выпадающего меню.

Настройка параметров клетки, группы клеток, объекта или всего шаблона рассмотрены в следующих темах:

- [Окно настройки свойств](#)
- [Настройка свойств клетки](#)
- [Макросы в клетках шаблона](#)
- [Настройка свойств группы клеток](#)
- [Настройка свойств объекта "Кнопка"](#)
- [Настройка свойств объекта "Надпись"](#)
- [Настройка свойств объекта "Флаг"](#)
- [Настройка свойств объекта "Переключатель"](#)
- [Настройка свойств объекта "Редактор"](#)
- [Настройка свойств объекта "Список"](#)
- [Настройка свойств объекта "Рисунок"](#)
- [Настройка свойств объекта "Ряд Закладок"](#)
- [Настройка свойств объекта "Проигрыватель"](#)
- [Настройка свойств объекта "Рамка"](#)
- [Настройка свойств объекта "OLEКонтейнер"](#)
- [Настройка свойств объекта "График"](#)
- [Настройка свойств объекта "КартотекаШаблона"](#)
- [Настройка свойств объекта "ДеревоКартотеки"](#)
- [Настройка свойств объекта "ПодтаблицаШаблона"](#)
- [Настройка свойств шаблона в целом](#)
- [Настройка свойств фрейма](#)
- [Настройка библиотеки стилей](#)

Если в свойствах клетки шаблона установлен флаг **Использовать макросы**, то в текст клетки можно включать специальные управляющие последовательности, которые позволяют переключать стиль, размер и цвет шрифта.

Большинство управляющих последовательностей записывается в угловых скобках <...>, аналогично тэгам html-разметки. Если символ "<" должен выводиться как обычно, следует написать "<<". Аналогичным образом удвоение символа ">>" означает, что необходимо его трактовать как обычный символ, а не как ограничитель макроса.

Поддерживаются следующие последовательности:

- <b+> - включить жирное начертание шрифта;
- <b-> - выключить жирное начертание шрифта;
- <b.> - сбросить жирность шрифта в исходное для клетки состояние;
- <i+> - включить наклонное начертание шрифта;
- <i-> - выключить наклонное начертание шрифта;
- <i.> - установить наклон шрифта в исходное для клетки состояние;
- <u+> - включить подчеркивание в начертании шрифта;
- <u-> - выключить подчеркивание в начертании шрифта;
- <u.> - установить признак подчеркивания в исходное для клетки состояние;
- <s+> - увеличить размер шрифта на 1 пункт;
- <s-> - уменьшить размер шрифта на 1 пункт;
- <s:nn> - установить размер шрифта в nn пунктов, например ;
- <s.> - установить исходный размер шрифта;
- <cf:xxx> - установить цвет шрифта, где xxx - числовое или мнемоническое обозначение цвета (см. далее);
- <cb:xxx> - установить цвет фона;
- <cf.> - сбросить цвет шрифта;
- <cb.> - сбросить цвет фона.

Для задания цвета возможно несколько вариантов:

<cf:\$BBGGRR> - задает цвет шрифта как шестнадцатеричное число, в формате RGB, то есть каждая из основных компонент цвета - красная, зеленая, синяя - кодируется числом от 0 (нет соответствующей цветовой компоненты) до 255 или \$FF (цветовая компонента имеет максимальную насыщенность); например:

\$FF0000 - Голубой;
\$00FF00 - Зеленый;
\$0000FF - Красный;
\$00FFFF - Желтый.

<cf:Цвет> - где, Цвет - мнемоническое обозначение цвета, например, <cf:White>.

Используются следующие обозначения для физических цветов:

White	(\$FFFFFF) // Белый
Black	(\$000000) // Черный
Red	(\$0000FF) // Красный
Lime	(\$00FF00) // Зеленый
Blue	(\$FF0000) // Голубой
Yellow	(\$00FFFF) // Желтый
Aqua	(\$FFFF00) // Бирюзовый
Fuchsia	(\$FF00FF) // Фиолетовый
Maroon	(\$000080) // Темно-красный
Green	(\$008000) // Темно-зеленый
Navy	(\$800000) // Темно-синий
Olive	(\$008080) // Темно-желтый
Teal	(\$808000) // Темно-бирюзовый
Purple	(\$800080) // Темно-фиолетовый
Gray	(\$808080) // Темно-серый
LtGray,Silver	(\$C0C0C0) // Светло-серый

Используются следующие обозначения для логических цветов из системной палитры Windows цветов:

Transparent // Прозрачный
AppWorkspace // Рабочая область
Window // Окно
WindowText // Текст окна
Highlight // Фон выбранного текста
HighlightText // Выбранный текст
BtnHighlight // Светлый кнопки
BtnFace // Поверхность кнопки, стандартный цвет диалога.
BtnShadow // Тень кнопки
BtnText // Текст кнопки
Menu // Меню
MenuText // Текст меню
GrayText // Запрещенный

При редактировании содержимого клетки последовательности вводятся как обычные символы (т.е. визуальное редактирование не поддерживается).

Последовательности могут использоваться не только в статических клетках, но и в полях ввода вывода, и в форматных преобразованиях. Например, можно задать [форматное преобразование](#), которое выводит отрицательные суммы красным цветом:

```
0.00; <cf:red>0.00<cf.> ; -
```


По команде **Библиотека стилей** (на панели инструментов "Шаблон" имеется соответствующая кнопка) на экране открывается окно настройки библиотеки стилей.

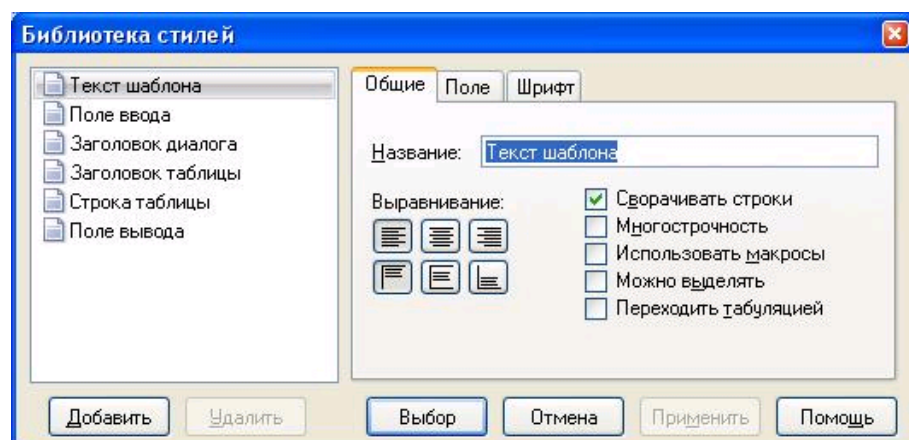


Рис. Библиотека стилей.

В левой части данного диалога перечисляются стили, входящие в библиотеку. С помощью кнопок действия **Добавить** и **Удалить** можно добавлять новые стили или удалять старые.

Правая часть диалога содержит информацию о настройках свойств клеток, использующих данный [стиль](#). По своему содержанию страницы "Общие", "Поле" и "Шрифт" аналогичны одноименным страницам [окна настройки свойств клетки](#).

Библиотека стилей сохраняется вместе с шаблоном бланка. Таким образом, различные бланки могут иметь разные по составу библиотеки стилей.

Изменение свойств группы клеток производится с помощью окна настройки свойств того же вида, что и для единственной [клетки](#).

Для проведения групповой операции по изменению свойств следует [выделить группу клеток](#) и открыть окно настройки свойств. В нем будут отображены только те параметры, которые имеют одинаковое значение у всех выделенных клеток. Остальные параметры не будут инициализированы.

Изменение того или иного параметра, в том числе и неинициализированного, приведет к его изменению у всех выделенных клеток.

Для отображения и настройки свойств клетки секции используется диалог "Свойства клеток", содержащий шесть страниц: "[Общие](#)", "[Формат](#)", "[Поле](#)", "[Шрифт](#)", "[Подсказка](#)", "[События](#)".

Страница "Общие"

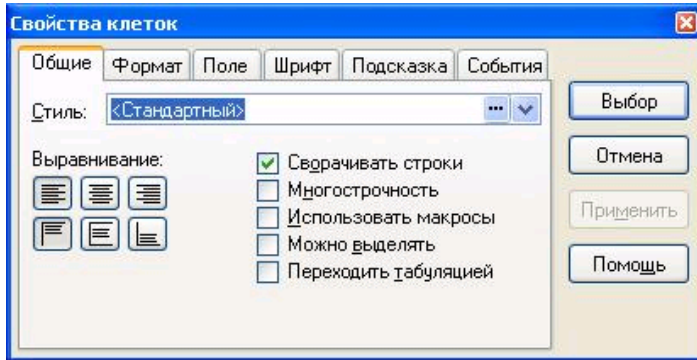



Рис. Страница "Общие" свойства клеток.

Поле **Стиль**

В поле указывается *стиль* текущей клетки, выбираемый из числа имеющихся в [библиотеке стилей](#). Кнопка  позволяет вызвать диалог "[Библиотека стилей](#)" и изменить имеющийся стиль оформления клетки или создать новый.

Группа кнопок **Выравнивание**

Тип выравнивания текста в клетке по центру, левому, правому, верхнему или нижнему краю задается путем выбора соответствующих кнопок.

Флаг **Сворачивать строки**

Позволяет включить или отключить режим [сворачивания длинных строк](#). По умолчанию данный флаг включен у каждой клетки, однако при большом числе клеток с таким свойством прорисовка шаблона может заметно замедлиться. В связи с этим рекомендуется снимать данный флаг у всех клеток и оставлять его включенным только там, где реально возможно появление длинных строк, не уступающих в клетках по ширине.

Флаг **Многострочность**

Установка флага позволяет представить текст в клетке в виде нескольких строк, т.е. разрешается переносить текст на следующую строку, нажатием клавиши *Enter* (перевод каретки). При снятом флаге переход на другую строчку невозможен.

Флаг **Использовать макросы**

Позволяет включить или отключить специальный механизм управления внешним видом клеток с помощью особого рода [макросов](#), записываемых непосредственно в самой клетке.

Флаг **Можно выделять**

По умолчанию данный флаг включен у каждой клетки. В этом случае данная клетка может быть выделена, т.е. на ней может быть установлен курсор и доступны команды контекстного меню.

Флаг **Переходить табуляцией**

Установка флага обеспечивает переход к данному полю с помощью клавиши **Tab**. В этом случае поле можно сделать активным, последовательно нажимая клавишу **Tab** до тех пор, пока фокус не переместится на него.

Страница "Формат"

Параметры, устанавливаемые на данной странице, задают тип и формат клетки, а также определяют, является ли данная клетка полем ввода или вывода.

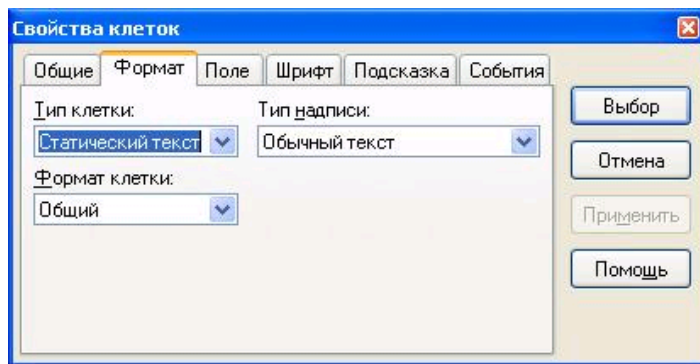



Рис. Страница "Формат" свойства клеток.

Флаг **Поле только для вывода**


Установка флага запрещает пользователю редактировать текущее поле, и делает его доступным только для просмотра. Ввод в поле осуществляется автоматически через программный интерфейс.

Флаг **Поле имеет кнопку выбора**

Когда данный флаг включен, у правого края текущей клетки столбца появляется , по нажатию которой будет формироваться событие [ПриОбзоре](#). Событие **ПриОбзоре** не генерируется, если поле доступно только на чтение (флаг **Поле только для вывода** включен) - в этом случае при нажатии на эту кнопку возникает событие [ПриНажатии](#) с параметром действия, равным *Шаблон.НажатиеКнопки* (*Template.ButtonPressed*). Если обработчик этого события отсутствует или в нем разрешено выполнение стандартной (встроенной в систему) обработки, то по умолчанию программа выполнит следующее стандартное действие:

- для полей типа **Дата** - открытие календаря;
- для числовых полей - открытие калькулятора;
- для ссылочных полей - открытие картотеки;
- для перечислимых полей - открытие списка выбора;
- для остальных полей - открытие списка истории.

Поле **Тип клетки**

Поле предназначено для выбора типа текущей клетки из предложенных вариантов, перечисленных в выпадающем списке, который открывается кнопкой .

- **Статический текст** - поле, содержащее обычный текст, кнопку, заголовок или гиперссылку. По умолчанию - это обычный текст. Допустимый тип клетки можно выбрать из выпадающего списка поля **Тип надписи**. Если для статического текста выбран тип кнопка, то в клетке шаблона будет фактически отображаться кнопка, которую пользователь может нажимать.
- **Поле ввода/вывода** - поле, в которое разрешен ввод или просмотр данных, его тип задается в поле **Формат клетки**;
- **Вычисляемое поле** - поле, ввод и вывод в которое осуществляются на прикладном уровне с помощью обработчиков событий, поле не связано ни с одной переменной. Вычисляемые поля предназначены для получения и отображения значений, вычисляемых, так сказать, "на лету". Например, в документе имеются два поля, в которых хранится цена товара и его количество. Стоимость товара всегда равна произведению количества на цену. Тогда логично сделать поле, в котором должна выводиться стоимость, вычисляемым, и задать формулу расчета в обработчике одного из событий (а именно, [ПриВыводе/OnOutput](#)). В результате, в бланке всегда будет отображаться сумма, соответствующая введенным цене и количеству.

Поле **Формат клетки**

В поле следует выбрать формат текущей клетки. Для каждого типа формата перечисляется набор интерфейсных элементов, связанный с ним. Ввод в поле производится из выпадающего списка, в котором приведены разрешенные форматы клеток:

- **Общий** - поле ввода-вывода без конкретизации типа. Программа автоматически присвоит полю тот тип, который имеет переменная в описании бланка.
- **Строка** - поле может быть использовано для ввода или вывода информации в переменные строкового типа. Для строковых полей разрешается в поле **Максимальное число символов** задать *ограничение на число символов, вводимых с клавиатуры при редактировании данной клетки*. Если поле **Максимальное число символов** не заполнено и текущая клетка в качестве поля содержит "поле записи", то максимальное количество символов, которые разрешено ввести, будет взято из MTL-описания данного поля.

Если в поле ввода необходимо при редактировании ввести перевод строки, то следует нажать комбинацию клавиш **Ctrl+Enter**.

- **Число** - поле используется для целых и числовых переменных. По умолчанию для числовых полей используется стандартный формат вывода чисел, которым в качестве разделителя целой и дробной части предусмотрено употреблять точку ("."). Наиболее употребительные форматы, которые удобно использовать для получения унифицированного способа отображения значений в клетке, перечислены в выпадающем списке поля **Формат вывода числа**. Внешний вид представления чисел в клетке для выбранного формата сразу же отображается в диалоге. Кроме того, можно также выбрать формат числа, в котором в качестве разделителя троек разрядов числа (триад) будет использоваться апостроф, например, 3'456.23. Допустимые символы форматного преобразования и их назначение рассматриваются в теме ["Преобразования формата"](#).
- **Дата** - используется в полях для задания дат. В этом случае также нужно выбрать формат их вывода в поле **Формат вывода даты**. По умолчанию задан стандартный формат вывода дат.
- **Логический** - поле логического типа, для этих полей поддерживается два варианта их представления: флаг и "залипающая" кнопка (с состояниями "нажата" и "отжата"). Выбор варианта производится с помощью поле **Формат отображения**, а заголовок кнопки/флага можно указать в поле **Заголовок**.
- **Перечислимый** - поле перечислимого типа, которое может быть связано с переменной целого или логического типа данных и списком строк, обозначающих альтернативные значения, которые указываются в поле **Варианты строк**.

Перечислимое поле позволяет пользователю (в режиме заполнения бланка, в сессии) выбрать нужное значение из заранее подготовленного списка, который может содержать две строки, если поле связано с логической переменной, или произвольное число строк, если в данное поле выводится переменная целого типа. При этом с каждой строкой из этого списка автоматически связывается значение переменной: при использовании логической переменной - это Ложь для первой строки в списке и Истина для второй. В случае связи перечислимого поля с целой переменной при выборе первого значения из списка в нее записывается 0, второго - 1 и т.д.

Например, в карточке, содержащей информацию о сотруднике, необходимо ввести его пол. Для этого в бланке можно описать переменную *МужскойПол* логического типа и связать ее с перечисляемым полем, имеющем в списке два варианта: "Женский" и "Мужской". В таком случае пользователь сможет выбрать необходимое значение из списка, а в переменную *МужскойПол* автоматически будет записано значение Ложь, если был выбран женский пол, или Истина, если пользователь предпочел второй вариант.

Если вариантов выбора больше, чем два, то в качестве переменной, хранящей результат ввода, следует использовать переменную целого типа. Например, в той же карточке следует указать подразделение, в котором работает сотрудник. Для этого пронумеруем все подразделения от нуля (0 - администрация, 1 - отдел маркетинга, 2 - производственный отдел и т.д.) и заведем в бланке переменную *НомерОтдела* для хранения этого номера. Тогда для ввода номера подразделения достаточно связать переменную *НомерОтдела* с перечислимым полем, в котором в списке вариантов будут записаны все подразделения в порядке их нумерации. При этом конечный пользователь будет работать только с названиями подразделений и ничего не знать об их внутренней нумерации, а сопоставление наименования отдела и его номера в переменной производится автоматически.

Установку нужного варианта в перечислимом поле можно производить, нажимая клавишу **Пробел** - по каждому нажатию вариант будет изменяться на следующий по списку. Кроме того, варианты можно выбирать, нажав кнопку **Enter** (для того чтобы перейти в режим редактирования поля) и затем - кнопку **PageDown**, в результате чего появляется выпадающий список со всеми альтернативными вариантами.

- **Ссылочный** - поле ссылочного типа, используемое для отображения ссылки на другой документ произвольного типа (этот тип задается при описании записи в MTL-файле). Однако в связи с тем, что ссылка представляет собой служебную информацию, мало понятную конечному пользователю, на экран, как правило, выводится не она, а содержимое поля другого документа, на который имеется ссылка.

Предположим, что в проекте существуют справочники накладных и контрагентов. Одно из полей накладной является ссылочным и используется для указания юридического лица, на которое выписывается накладная. Для того чтобы в ссылочном поле в сессии отображалось понятное пользователю название контрагента достаточно в режиме разработки в ссылочном поле после названия типа документа и точки указать имя свойства документа, например, "#Контрагент.КорИмя".

Поля и флаги, отображаемые на странице "Формат" для ссылочных полей

Поле **Картотека**

В поле **Картотека** указывается имя картотеки, на которую хранится ссылка в текущем ссылочном поле. При попытке отредактировать значение поля картотека автоматически открывается, если флаг **Разрешить ручной ввод** снят.

Флаг **Разрешить ручной ввод**

Если у ссылочного поля установлен флаг **Разрешить ручной ввод**, то при входе в поле появляется inplace-

редактор, в котором можно набрать некоторый текст. При этом, если ссылка (идентификатор после символа '#') содержит только тип документа без уточняющего имени свойства, имеющегося у данного типа (напомним, что свойства можно указывать после имени класса документа через точку), то набранный текст интерпретируется как DocID (в формате "{DocumentClassName:DocID}", например, "{Накладная:56473}"). При этом поиск документа среди документов указанного типа выполняется по полю DocID. Если же ссылка включает имя какого-либо свойства документа, указанное через точку после имени типа документа, (например, "Накладная.Номер"), то набранный в поле ввода текст используется для поиска документа по данному полю (колонке) внутри картотеки (связанной с полем). При точном совпадении найденный документ автоматически присваивается ссылочной переменной, в противном случае открывается картотека и позиционируется на ближайший похожий документ.

Поля **Поле** и **Фильтр**

Разработчик имеет возможность явно указать поле документа, по которому будет проводиться обзор в картотеке, не только записав его имя в ссылке, но и через диалог свойств клетки - для этого достаточно ввести имя поля в поле ввода **Поле** (поле обзора). При этом обзор будет проводиться по указанной колонке картотеки по всем записям, удовлетворяющим условию фильтра, которое записано в поле **Фильтр** даже в том случае, когда в клетке шаблона записано другое свойство записи. Например, если в клетке стоит "#Контрагент.Название", а в диалоге свойств установлен обзор по полю "Адрес", то записи в картотеке будут просматриваться именно по адресу, а не по названию контрагента, однако в клетке по-прежнему будет отображаться название. Вместе с тем, когда в клетке открывается inplace-редактор, в нем отображается значение именно из "обзорного" поля (если оно было указано). Поле ввода **Поле** имеет смысл заполнять только для клеток, в которых разрешен ручной ввод.

Быстрый поиск при редактировании поля

Когда на стадии выполнения проекта в ссылочном поле открыт inplace-редактор (т.е. в данном поле разрешен ручной ввод, и пользователь нажал **Enter** или выполнил щелчок мышью в данном поле), в нём работает так называемый *круизинг* (*быстрый поиск*). Заключается он в том, что по нажатию специальных горячих клавиш (см. ниже) происходит перемещение по набору записей, доступных для ввода в данное поле, и значение ссылки в данном поле изменяется. Круизинг можно себе представить таким образом, как будто для редактируемого поля открыта невидимая картотека, и пользователь перемещает курсор по этой картотеке, а каждая вновь выделенная в ней запись отображается в поле ввода. Записи в этой "невидимой картотеке" упорядочены по полю обзора (если оно задано в поле ввода **Поле**) или по полю разыменования (имя, указанное через точку после идентификатора ссылочной переменной). Если не задано ни обзорного поля, ни разыменования, круизинг невозможен.

Существуют следующие варианты перемещения:

- Ctrl+стрелка вправо - переход на следующую запись;
- Ctrl+стрелка влево - переход на предыдущую запись;
- Ctrl+Home - переход на первую запись;
- Ctrl+End - переход на последнюю запись.

Дополнительно, если установлен обработчик события [ПриНаборе/OnType](#) то поиск документа (и заполнение ссылочного поля), удовлетворяющего набранному тексту, может быть осуществлен программно внутри обработчика.

Поля **Поле** и **Фильтр**

Разработчик имеет возможность явно указать поле документа, по которому будет проводиться обзор в картотеке, не только записав его имя в ссылке, но и через диалог свойств клетки - для этого достаточно ввести имя поля в поле ввода **Поле** (поле обзора). При этом обзор будет проводиться по указанной колонке картотеки по всем записям, удовлетворяющим условию фильтра, которое записано в поле **Фильтр** даже в том случае, когда в клетке шаблона записано другое свойство записи. Например, если в клетке стоит "#Контрагент.Название", а в диалоге свойств установлен обзор по полю "Адрес", то записи в картотеке будут просматриваться именно по адресу, а не по названию контрагента, однако в клетке по-прежнему будет отображаться название. Вместе с тем, когда в клетке открывается inplace-редактор, в нем отображается значение именно из "обзорного" поля (если оно было указано). Поле ввода **Поле** имеет смысл заполнять только для клеток, в которых разрешен ручной ввод.

Страница "Поле"

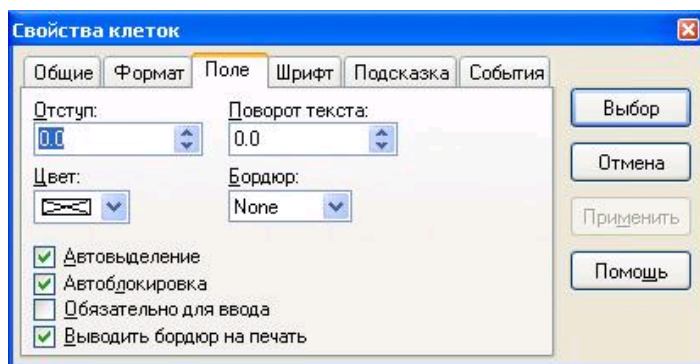


Рис. Страница "Поле" свойств клеток.

Поле **Отступ**

В данном поле указывается размер зазора (в мм) между внешней границей клетки и ее внутренней рабочей областью, где выводятся/вводятся данные и текст. Отступ одинаков для всех 4-х сторон клетки. Если отступ больше 0, то рабочая область клетки (поле) окружена рамкой, сквозь которую "проглядывает" фон клетки. Бордюр, придающий клетке трехмерный вид, выводится по периметру рабочей области, заданной отступами.

Поле **Цвет**

Позволяет назначить цвет для самого поля - этот цвет может отличаться от цвета клетки шаблона, в котором данное поле расположено.

Поле **Поворот текста**

Данная опция позволяет поворачивать текст внутри клетки на фиксированные углы. Ввод в поле производится кнопкой . Если задано значение 0 (градусов), то текст поворачиваться не будет. Поворот текста учитывает способы выравнивания текста по горизонтали и вертикали. Если текст повернут на 90 или - 90, то такие клетки не участвуют в определении авто высоты строки секции.

Примечание. Допускаются следующие значения углов поворота текста -180.0, -90.0, 0.0, 90.0, 180.0. Если провести условную горизонталь через середину клетки шаблона и указать положительное направление вправо, то угол поворота против часовой стрелки считается положительным.

Поле **Бордюр**

В поле вводится только разрешенное значение, указанное в выпадающем списке (открывается кнопкой). При выборе значения None бордюр вокруг клетки отсутствует, В остальных случаях прорисовка бордюра по периметру рабочей области клетки выполняется одним из выбранных способов: по умолчанию (Default), тонкий (Single), статический (Static), внутренний (Client) или оконный (Window) бордюр.

Флаг **Автовыделение**

Данная опция позволяет выделять текст (флаг установлен) при входе в клетку (при открытии встроенного in-place редактора). Если флаг снят, то текст при входе в текущую клетку выделен не будет, что позволяет избежать случайного удаления выделенного текста.

Флаг **Автоблокировка**

Данная опция имеет смысл только для полей бланка-редактора. По умолчанию флаг установлен, т.е. документ автоматически переводится в режим редактирования при входе в поле. При снятом флаге документ будет переведен в режим редактирования при изменении значения при завершении редактирования в поле.

Страница "Шрифт"

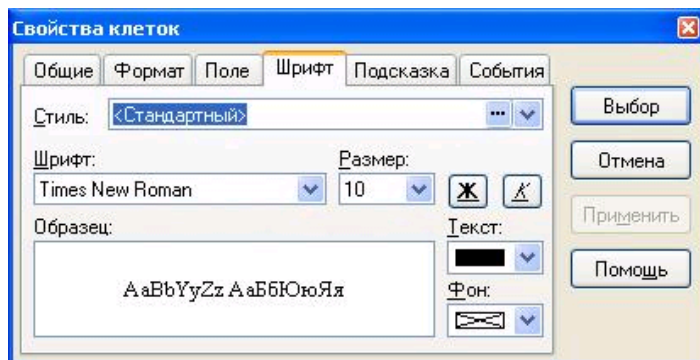


Рис. Страница "Шрифт" свойств клеток.

Страница "Шрифт" служит для установки *параметров шрифта* в текущей клетке. К ним относятся стиль клетки, выбранный из [библиотеки стилей](#), шрифт, которым будет выводиться текст в клетке, способ начертания (обычный, курсив, жирный), а также цвета фона и текста. В поле **Образец** можно просмотреть, как будет выглядеть текст в текущей клетки с учетом всех сделанных установок.

Страница "Подсказка"

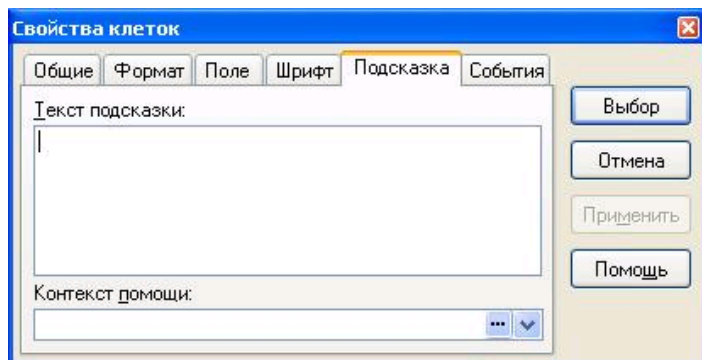


Рис. Страница "Подсказка" свойств клеток.

Поле **Текст подсказки**

В поле вводится произвольный текст, поясняющий назначение текущей клетки, который будет отображаться в специальном всплывающем окне, если навести курсор на это поле.

Поле **Контекст помощи**

Если требуется более подробная информация, то в этом поле можно также указать путь к файлу с текстом помощи. Этот текст будет появляться в специальном всплывающем окне, если нажата клавиша **F1** и фокус установлен на текущем поле.

Страница "События"

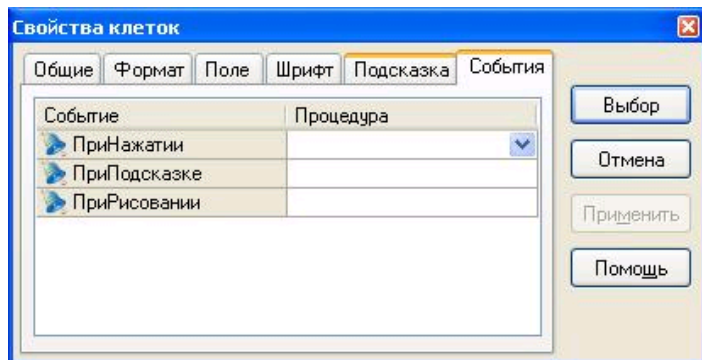


Рис. Страница "События" свойств клеток.

Эта страница содержит список [событий](#), предусмотренных для клетки данного типа, и имена методов-обработчиков.

Список событий, предусмотренных для полей бланка, и требования к их обработчикам приведены в разделе, посвященном объекту [КлеткаШаблона](#).

Настройка свойств объекта "OLEКонтейнер"

Объект "OLEКонтейнер" добавляется в шаблон бланка с помощью инструментальной кнопки **Вставить объект | OLEКонтейнер** и может содержать произвольный OLE-документ.

Для отображения и изменения характеристик OLEКонтейнера используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены четыре страницы: "Общие", "Положение", "Подсказка" и "События".

Страницы "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для OLEКонтейнера.

Страница "Общие" позволяет задать *название объекта-OLEКонтейнера*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "OLEКонтейнер"](#). Кроме того, данная страница имеет поле ввода Переменная, где можно указать имя переменной типа **OLEДокумент** или поля записи того же типа. В этом случае все манипуляции с объектом-контейнером будут приводить к синхронному изменению переменной или записи информационной базы.

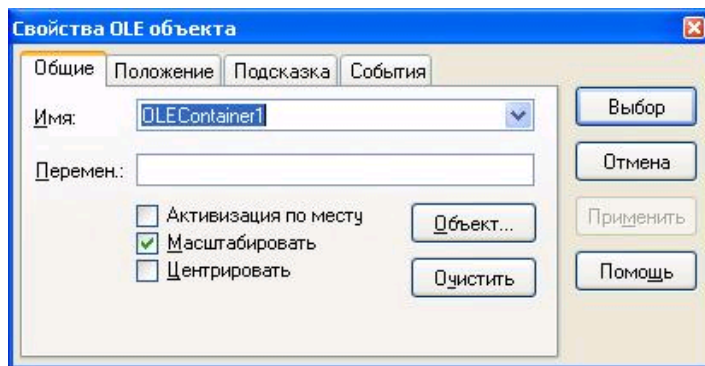


Рис. Страница "Общие" свойств объекта "OLEКонтейнер".

Конкретный OLE-документ легко назначить непосредственно во время разработки шаблона - для этого предназначена кнопка "Объект".

По нажатию кнопки **Очистить** имеющийся в контейнере документ удаляется из памяти.

Флаг **Активизация по месту** позволяет выбрать режим редактирования OLE-документа, помещенного в OLE-контейнер. Когда флаг включен, документ открывается непосредственно внутри окна контейнера, а в противном случае - в окне приложения, создавшего документ.

Если установлен флаг **Масштабировать**, то изображение OLE-документа растягивается или сжимается под размер контейнера, в зависимости от их соотношения. Иначе документ отрисовывается в объекте-контейнере в натуральную величину.

Флаг **Центрировать** определяет, нужно ли центрировать изображение документа внутри контейнера. Данный флаг имеет смысл только в том случае, если отключено масштабирование.

Объект "График" добавляется в шаблон бланка с помощью инструментальной кнопки **Вставить объект | График** и предназначен для визуализации наборов данных.

Для отображения и изменения характеристик графика используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены четыре страницы: "Общие", "Положение", "Подсказка" и "События".

Страницы "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических для графика.

Страница "Общие" позволяет задать *название объекта-графика*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "График"](#).

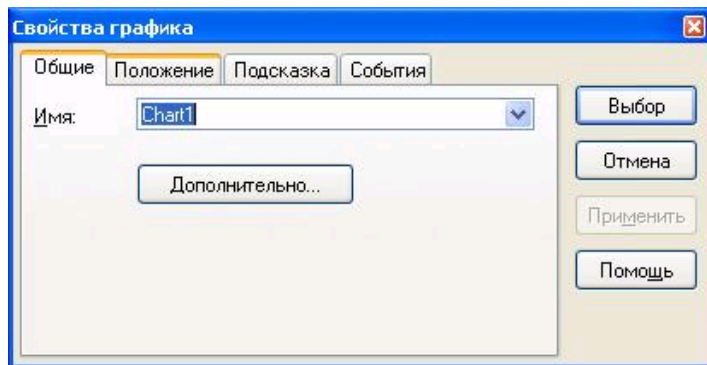


Рис. Страница "Общие" свойств графика.

Также на странице "Общие" расположены элементы управления, позволяющие изменять внешний вид графика и определять тип выводимых на него показателей (специфические термины, используемые при настройке графика и связанные с принципами его отображения, а также элементы пользовательского интерфейса, которые описываются в [отдельном разделе](#)).

Из выпадающего списка **Вид графика** можно выбрать внешнее представление данных. Существуют следующие варианты:

- Горизонтальная гистограмма;
- Вертикальная гистограмма;
- Заполненная диаграмма;
- График (кусочно-линейный);
- Точечная диаграмма;
- Круговая диаграмма;
- Стрелочная диаграмма;
- Пузырьковая диаграмма.

Флаг **Легенда** позволяет включать и отключать отображение легенды (легенда определяет соответствие между цветом элементов графика и их принадлежностью к различным сериям данных). Если он включен, то доступен выпадающий список **Разместить**, где перечислены варианты размещения легенды: слева, справа, снизу или сверху от графика.

При включенном флаге **Трехмерный вид** все элементы графика выводятся объемными.

Сетку на графике можно включать или отключать с помощью флагов **Горизонтальная сетка** и **Вертикальная сетка**.

При необходимости изображение график может быть отображено перевернутым, для чего предназначен флаг **Перевернут график**.

Флаг **Подсказки** управляет видимостью выносных элементов на графике, в которых выводятся значения величин.

Флаг **Штриховка** переключает график между режимами отображения сплошным цветом и со штриховкой.

По нажатию кнопки **Дополнительно**, разработчик может вызвать [диалог "Настройки графика"](#).

На странице "События" приводится список событий, генерируемых программой в ответ на действия пользователя. В левой колонке перечня приведены названия событий, а в правой колонке разработчик может задать программные обработчики соответствующих событий. Принцип действия страницы "События"

аналогичен таким же [страницам](#) в диалогах свойств объектов шаблона.

По нажатию кнопки **Выбор** сделанные изменения вступают в силу и диалог закрывается.

По нажатию кнопки **Применить** сделанные изменения вступают в силу, но диалог остается открытым.

Кнопка **Отмена** позволяет закрыть диалог без сохранения изменений.

Настройка свойств объекта "ДеревоКартотеки"

Объект "ДеревоКартотеки" добавляется в шаблон бланка с помощью инструментальной кнопки **Вставить объект | ДеревоКартотеки** и позволяет вставить в шаблон окно с древовидным представлением иерархии групп.

Для отображения и изменения характеристик картотеки используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположено пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для дерева картотеки.

На странице "Общие" в поле **Имя** указывается идентификатор объекта дерева.

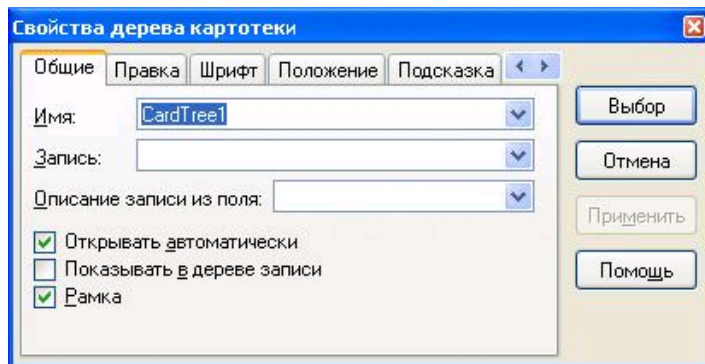


Рис. Страница "Общие" свойств дерева картотеки.

Поле **Запись**, снабженное выпадающим списком, позволяет выбрать имя класса записи, связанной с картотекой. В списке перечисляются все иерархические классы записей проекта. Для гетерогенной картотеки из списка можно выбрать более 1 класса записи - для этого необходимо включить флажки слева от имен классов в требуемых элементах списка.

Далее в диалоге свойств идет **Описание записи из поля**. Значения указанного здесь поля записи будут выводиться в качестве заголовков групп в дереве. Для заполнения поля рекомендуется использовать выпадающий список, в котором перечисляются все поля класса записи (записей), введенного в поле **Запись**.

Флаг **Открывать автоматически**, будучи включенным, предписывает системе автоматически заполнять дерево данными из информационной базы. Если флаг сброшен, дерево необходимо заполнять данными программно.

Флаг **Показывать удаленные записи** позволяет разрешить или запретить отображение в дереве групповых записей, помеченных удаленными.

Страница "Правка" в целом аналогична такой же странице для объекта [КартотекаШаблона](#), но не имеет флага **Редактировать в клетке** и поля ввода бланка-редактора **Для записи**.

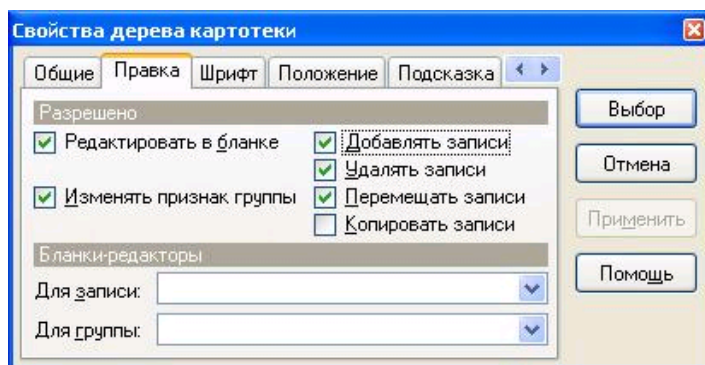


Рис. Страница "Правка" свойств дерева картотеки.

Объект "КартотекаШаблона" добавляется в шаблон бланка с помощью инструментальной кнопки **Вставить объект | КартотекаШаблона** и позволяет внедрить в шаблон табличную часть картотеки (для иерархических картотек дополнительно можно также вставить в шаблон и объект ["ДеревоКартотеки"](#)).

Для отображения и изменения характеристик картотеки используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположено шесть страниц: ["Общие"](#), ["Правка"](#), "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно так же, как и для [окна настройки свойств](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для объекта "КартотекаШаблона".

Страница "Общие"

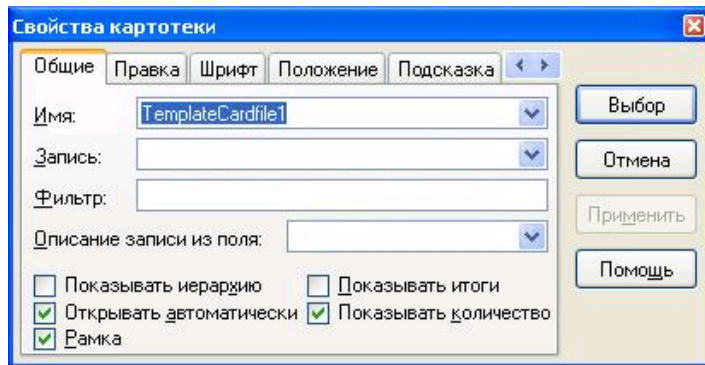


Рис. Страница "Общие" свойств картотеки шаблона.

Поля **Имя** и **Запись**

В полях указывается соответственно идентификатор объекта картотеки и имя класса записи, связанной с картотекой. Поля, снабжены выпадающими списками, обеспечивающими ввод в эти поля. Список поля **Запись** содержит все имеющиеся в проекте записи. Для гетерогенной картотеки можно выбрать более одного класса записи - для этого необходимо включить флаги слева от имен классов в требуемых элементах списка.

Поле **Фильтр**

При открытии картотека будет содержать только записи, удовлетворяющие установленному фильтру, т.е. логическому выражению, введенному в это поле.

Поле **Описание записи из поля**

Данное поле предназначено для иерархических картотек. Значения указанного здесь поля записи будут выводиться в качестве заголовков групп в дереве. Если поле не заполнять, то по умолчанию в качестве имен групп берутся **DocID** соответствующих групповых записей. Кроме того, это же поле для всех картотек служит источником обозначения записи, которое выводится пользователю при попытке выполнить над записью операцию, требующую подтверждения. В этом случае описание записи (т.е. содержимое указанного поля записи) выводится в диалоговом окне подтверждения.

Для заполнения поля рекомендуется использовать выпадающий список, в котором перечисляются все поля класса записи (записей), введенного в поле **Запись**.

Флаг **Показывать иерархию**

Флаг определяет, следует ли отображать записи картотеки в иерархическом виде (только элементы текущей группы) или в "плоском" (все записи картотеки). Если флаг включен, в каждый момент времени окно картотеки содержит записи текущего уровня иерархии.

Флаг **Открывать автоматически**

Установка флага предписывает системе заполнять объект "КартотекаШаблона" данными из информационной базы автоматически. Если флаг снят, объект необходимо заполнять данными программно.

Флаг **Показывать итоги**

При установке флага в картотеке шаблона показывается итоговая строка, иначе - строка не видна.

Флаг **Показывать количество**

Установка флага разрешает показывать количество записей в картотеке, при снятом флаге показ запрещен.

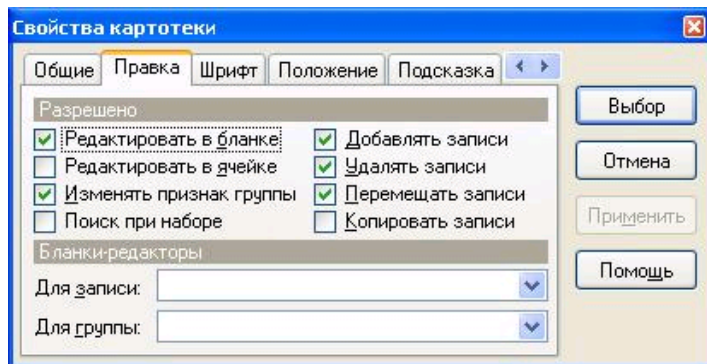


Рис. Страница "Правка" свойств картотеки шаблона.

Флаг **Редактировать в бланке**

Включение флага разрешает открывать бланк-редактора для ввода и изменения записей картотеки. Сам бланк-редактор (для одиночной записи и для группы записей) назначается с помощью полей ввода в группе **Бланки-редакторы** (см. далее).

Флаг **Редактировать в ячейке**

Флаг позволяет разрешить или запретить редактирование данных непосредственно в ячейках таблицы картотеки. Когда флаг включен, редактирование разрешено.

Флаг **Изменять признак в группе**

При включенном флаге пользователю разрешается изменять статус записи, т.е. групповая запись может стать простой и наоборот. При снятом флаге статус записи изменить нельзя.

Флаг **Поиск при наборе**

Если флаг включен, то в режиме редактирования (inplace-редактор) при нажатии на любую клавишу открывается диалог поиска в картотеке.

Внимание. Если текущий документ (картотека) уже находится в состоянии редактирования, то флаг игнорируется.

Флаги **Добавлять записи**, **Удалять записи**, **Перемещать записи** и **Копировать записи**

Флаги позволяют включить или запретить выполнение операций добавления, удаления, перемещения и копирования записей в картотеке. Если соответствующий флаг установлен, то операция разрешена. Под перемещением понимается перенос записи из одной группы в другую методом drag'n'drop или с помощью команд вырезания и вставки.

Группа полей **Бланки-редакторы**

Эти поля позволяют назначить для картотеки бланки-редакторы, то есть бланки, которые будут вызываться из картотеки при попытке отредактировать какой-либо документ. При необходимости оба поля могут быть заполнены из выпадающих списков этих полей или оставлены пустыми, или же может быть назначен лишь один бланк-редактор. Например, если для группы выбран конкретный класс бланка-редактора, то именно этот бланк будет использоваться для редактирования документов, которые описывают (представляют собой) группу.

Объект "Кнопка" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Кнопка**.

Для отображения и изменения параметров кнопки используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно так же, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для кнопки.

Страница "Общие"

Страница "Общие" позволяет задать общие свойства объекта "Кнопка", такие как имя, надпись на кнопке, способ ее выравнивания и другие свойства.

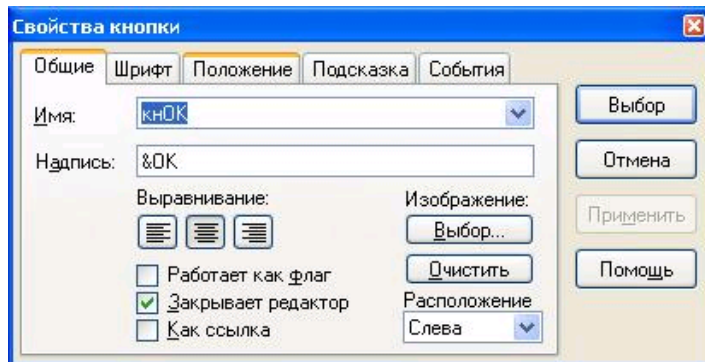


Рис. Страница "Общие" объекта кнопка.

Поля **Имя**, **Надпись**

В первом поле указывается *имя (идентификатор)* объекта "Кнопка", под которым объект будет известен бланку и может использоваться в качестве [переменной типа "Кнопка"](#), во втором поле - надпись, выводимую на кнопку. **Имя** Кнопка ▾ поля открывает список всех объектов, размещенных на текущем шаблоне, с указанием их типов. При выборе того или иного объекта из этого списка он активизируется, а окно свойств начинает отображать его параметры.

Кнопки **Выравнивание**

Данные кнопки определяют, каким образом будет выровнена надпись и рисунок (см. далее) на кнопке: по левому краю, по центру или по правому краю. Способ выравнивания задается нажатой кнопкой. Только одна из кнопок этой группы может быть нажата, поэтому при нажатии какой-либо кнопки, та кнопка, которая была в нажатом состоянии, возвращается в отжатое положение.

Флаг **Работает как флаг**

Флаг позволяет включить режим, когда кнопка может находиться в двух состояниях: нажатом и отжатом (нормальном). Если данный флаг включен, система по каждому нажатию кнопки меняет ее состояние на противоположное. Для того чтобы узнать или изменить состояние кнопки из программы ТБ.Скрипт используется свойство класса **Кнопка** - [Состояние](#).

Флаг **Закрывает редактор**

По умолчанию флаг установлен. При нажатии на любую кнопку шаблона автоматически закрывается редактор поля, чтобы этого не происходило флаг следует снять. При снятом флаге можно воспользоваться свойством [ТекущийРедактор](#).

Кнопки **Выбор** и **Очистить**

На объекте "Кнопка" может выводиться изображение. Для его назначения необходимо нажать кнопку **Выбор** и выбрать графический файл одного из поддерживаемых типов (JPEG, BMP, ICO, EMF, WMF). Для того чтобы убрать картинку с объекта "Кнопка", необходимо нажать кнопку **Очистить**. Надпись и изображение на кнопке могут выводиться одновременно - при этом они отображаются рядом. Возможно также задавать только надпись или только изображение.

Объект "Надпись" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Надпись**.

Для отображения и изменения характеристик надписи используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для надписи.

Страница "Общие" позволяет задать *название объекта-надписи*, под которым она будет известна бланку и может использоваться в коде в качестве [переменной типа "Надпись"](#), а также надпись, то есть текст выводимый внутри объекта Надпись. Также на этой странице можно задать тип выравнивания надписи: влево, вправо, по центру.

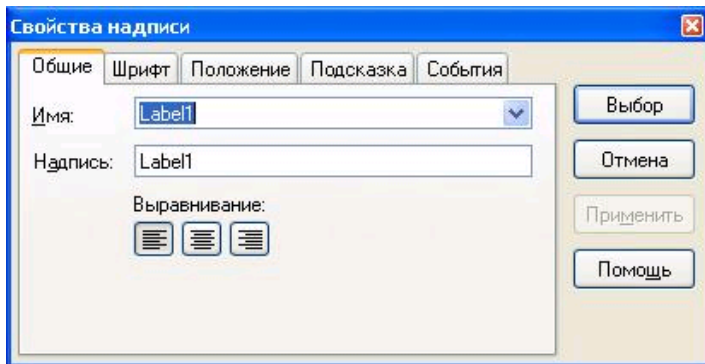


Рис. Страница "Общие" свойств надписи.

Объект "Переключатель" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Переключатель**.

Для отображения и изменения характеристик переключателя используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположено пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для переключателя.

Страница "Общие" позволяет задать *название переключателя*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Переключатель"](#), а также *надпись*, выводимую справа от переключателя.

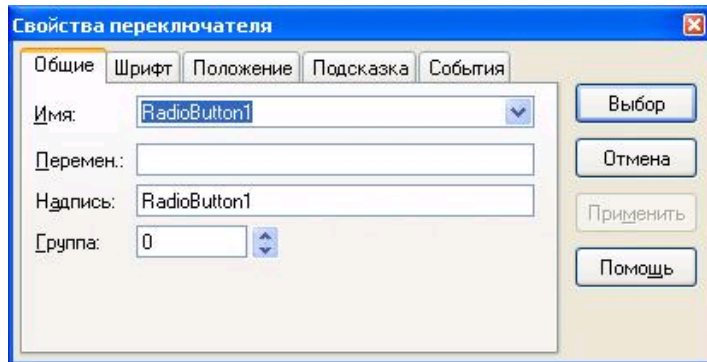


Рис. Страница "Общие" свойств переключателя.

В дополнение к перечисленным параметрам, страница "Общие" содержит поле ввода **Группа**.

Переключатели, у которых номер группы один и тот же, считаются взаимоисключающими; среди них только один может быть установлен. Объединение переключателей в группы используется для выбора одного варианта из нескольких.

Имеет смысл связать несколько переключателей, объединенных в группу, с одной переменной целого типа, используя поле **Перемен.** на странице "Общие". В этом случае при выборе одного переключателя из нескольких в эту переменную будет автоматически записываться номер варианта, определяемый взаимным расположением переключателей (вернее, последовательностью их добавления на шаблон).

Например, для выбора номера квартала можно расположить друг над другом четыре переключателя с надписями "I квартал", "II квартал", "III квартал" и "IV квартал", объединить их, установив для каждого один и тот же номер группы, и связать с одной и той же переменной, например, **Квартал**. Тогда при выборе первого - самого верхнего - варианта в переменную **Квартал** будет записано значение 0, второго - 1 и т.д.

В принципе объект "Переключатель" можно связать не только с целочисленной переменной, но и переменной типов логическое и перечисление ([type](#)). При этом логическая переменная может быть связана лишь с группой из двух переключателей: первый будет соответствовать значению FALSE, а второй - TRUE. Для правильной работы переключателя с перечислением, последнее должно иметь элементы со значениями от 0 до числа переключателей минус 1 (причем без пропусков), например, следующее перечисление:

```
type VSex = (varNonApplicable = 0, varMale, varFemale);
```

пригодно для группы из трех переключателей.

Настройка свойств объекта "ПодтаблицаШаблона"

Объект "ПодтаблицаШаблона" добавляется в шаблон бланка с помощью инструментальной кнопки **Вставить объект | ПодтаблицаШаблона** и позволяет внедрить в шаблон таблицу для отображения содержимого многозначных структурных полей записей. Данный объект как правило дополняет объект **КартотекаШаблона**.

Для отображения и изменения характеристик картотеки используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположено шесть страниц: "Общие", "Правка", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для картотеки.

На странице "Общие" в поле **Имя** указывается идентификатор объекта картотеки.

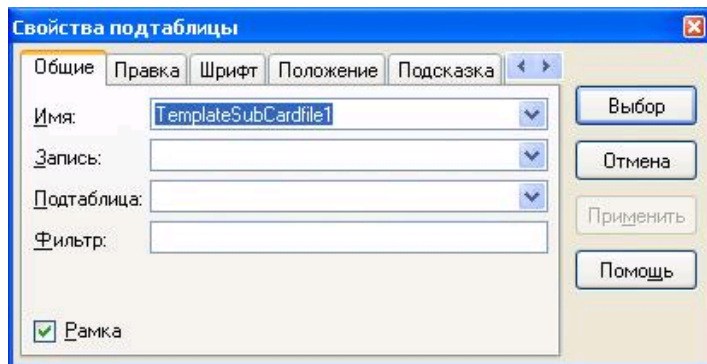


Рис. Страница "Общие" свойств подтаблицы шаблона.

Поле **Запись**, снабженное выпадающим списком, позволяет ввести имя класса записи, связанной с картотекой. Список содержит все имеющиеся в проекте записи. Для гетерогенной картотеки можно выбрать более 1 класса записи - для этого необходимо включить флажки слева от имен классов в требуемых элементах списка.

Поле **Подтаблица** с выпадающим списком позволяет выбрать одну из имеющихся в записи подтаблиц для отображения в объекте **ПодтаблицаШаблона**.

Поле **Фильтр** позволяет задать строковое выражение логического типа для отбора записей из подтаблицы, которые следует отображать в объекте: выводятся только те записи подтаблицы, для которых истинно выражение фильтра.

Флаг **Открывать автоматически**, будучи включенным, предписывает системе заполнять картотечный объект данными из информационной базы автоматически. Если флаг сброшен, объект картотеки необходимо заполнять данными программно.

На странице "Правка" размещена группа флагов, влияющая на поведение объекта.

Флаг **Редактировать в ячейке** позволяет разрешить или запретить редактирование данных непосредственно в ячейках подтаблицы.

Расположенные в правой половине страницы флаги позволяют разрешить следующие действия: **Добавлять строки**, **Удалять строки**, **Перемещать строки**.

Настройка свойств объекта "Проигрыватель"

Объект "Проигрыватель" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Проигрыватель** и служит для воспроизведения мультимедиа-файлов.

Для отображения и изменения характеристик проигрывателя используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены четыре страницы: "Общие", "Положение", "Подсказка" и "События".

Страницы "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для проигрывателя.

Страница "Общие" позволяет задать *название объекта-проигрывателя*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Проигрыватель"](#).

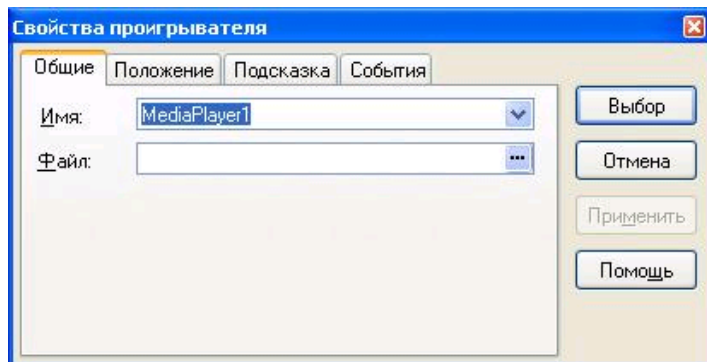


Рис. Страница "Общие" свойств проигрывателя.

Кроме того, данная страница содержит поле для указания мультимедиа-файла, который следует воспроизвести. Справа от поля расположена кнопка, при нажатии на которую открывается диалог выбора файла.

Для запуска проигрывателя и его остановки объект "Проигрыватель" обладает методами [Play](#) и [Stop](#). Пример использования этих методов приведен в разделе [Управление элементами шаблона из процедур и функций](#).

Настройка свойств объекта "Рамка"

Объект "Рамка" служит для визуального объединения группы объектов на шаблоне при оформлении диалоговых бланков и может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Рамка**.

Для отображения и изменения характеристик рамки используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для рамки.

Страница "Общие" позволяет задать *название объекта-рамки*, под которым она будет известна бланку и может использоваться в коде в качестве [переменной типа "Рамка"](#).

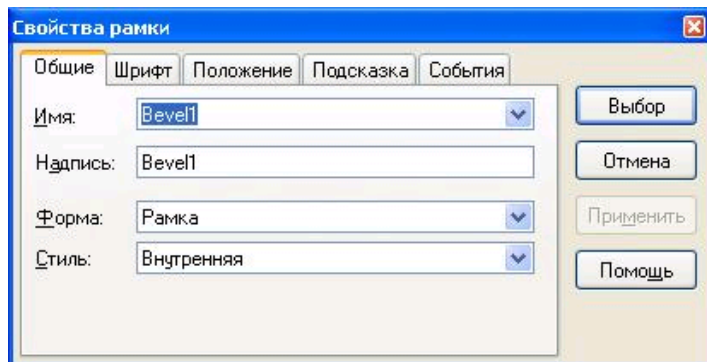


Рис. Страница "Общие" свойств рамки.

Кроме того, данная страница содержит поля для выбора формы (рамка/панель/вертикальная линия/горизонтальная линия) и стиля (внутренняя/внешняя) объекта "Рамка".

Следует отметить, что *надпись выводится* только в том случае, если выбрана форма "рамка".

Объект "Редактор" (поле ввода) может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Редактор**.

Для отображения и изменения характеристик редактора используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для редактора.

Страница "Общие" позволяет задать *название объекта-редактора*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Редактор"](#), а также тип выравнивания текста внутри редактора: влево, вправо, по центру.

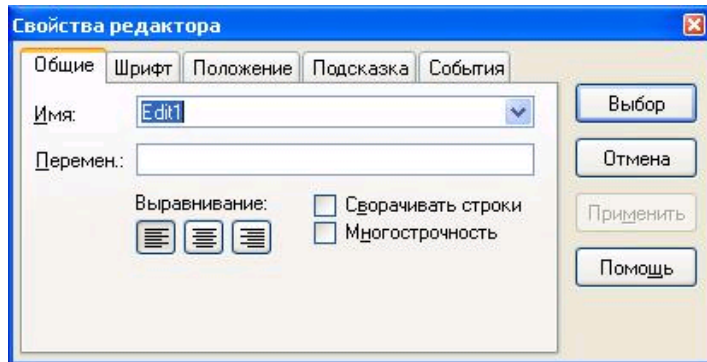


Рис. Страница "Общие" свойств редактора.

Флаг **Сворачивать строки** задает свойство редактора, позволяющее переносить слова на следующие строки, если текст не умещается на одной строке. Если флаг отключен (это положение по умолчанию), набираемый в редакторе текст прокручивается по горизонтали (и остается в виде единой строки). Если флаг включен, то редактор фактически становится многострочным.

Объект "Редактор" может быть связан с переменной любого типа для ввода в нее значения. Для этого достаточно указать ее имя в поле **Перемен.** на странице "Общие".

Редакторы, связанные с переменной типа "Ссылка на запись", позволяют только просматривать значение ссылки, но не редактировать его.

Объект "Рисунок" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Рисунок**.

Для отображения и изменения характеристик рисунка используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены четыре страницы: "Общие", "Положение", "Подсказка" и "События".

Страницы "Положение", "Подсказка" и "События" выглядят и действуют точно так же, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для рисунка.

Страница "Общие" позволяет задать *название объекта-рисунка*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Рисунок"](#).

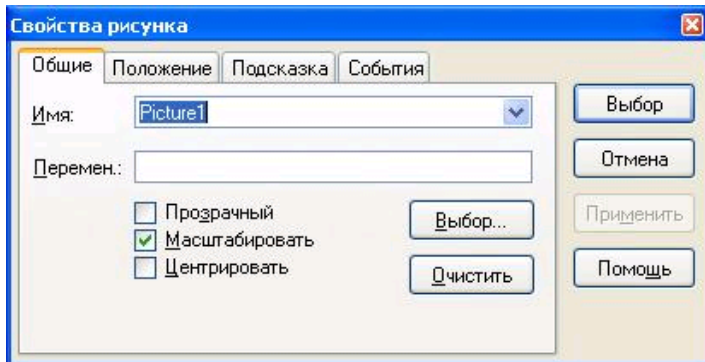


Рис. Страница "Общие" свойств рисунка.

Кроме того, данная страница содержит кнопку **Выбор**. При нажатии данной кнопки открывается диалог выбора файла, с помощью которого можно назначить файл с рисунком для отображения в бланке.

Рисунок можно связать с переменной типа "Изображение" (то есть, объектом класса Изображение), указав имя переменной в соответствующем поле. Если это сделать, то любая модификация переменной приведет к изменению того, что показывает рисунок и наоборот - любая модификация рисунка приведет к изменению того, что хранит переменная типа "Изображение". Иными словами, если объект шаблона "Рисунок" связан с переменной типа "Изображение", то он всегда показывает ее содержимое, а любой файл, подключенный к рисунку в результате нажатия кнопки **Выбор**, игнорируется.

Аналогично, в бланке-редакторе возможно установить взаимосвязь объекта "Рисунок" с полем типа "Изображение". Это дает возможность загружать в информационную базу изображения, просматривать их, заменять на другие, сохранять во внешний файл и так далее.

У рисунка есть всплывающее меню (выводится по нажатию правой кнопки мыши на рисунке), с помощью которого удобно модифицировать рисунок.

Пункты меню и их назначение:

- **Очистить** - удаляет текущее изображение из рисунка;
- **Копировать** - копирует изображение в буфер обмена;
- **Вставить** - вставляет изображение из буфера обмена в объект "Рисунок" (и связанную с ним переменную);
- **Загрузить из файла** - загружает картинку из файла в объект "Рисунок" (и связанную с ним переменную);
- **Сохранить в файл** - сохраняет изображение в файл.

Полезным для разработчика может оказаться флаг **Масштабировать**. Если данный флаг установлен, то изображение растягивается или сжимается под размер рисунка, в зависимости от их соотношения. Иначе изображение отрисовывается в объекте типа **Рисунок** в натуральную величину, и при этом, если оно не помещается, то появляются полосы прокрутки.

Если установлен флаг **Прозрачный**, то фон рисунка становится прозрачным за счет специальной предобработки изображения перед отрисовкой: цвет первой точки изображения считается прозрачным, и все фрагменты данного цвета не выводятся, открывая поверхность нижележащего шаблона. В противном случае изображение выводится в том виде, как есть, полностью перекрывая прямоугольный участок шаблона.

Флаг **Центрировать**, когда включен, предписывает центрировать изображения внутри объекта "Рисунок".

Объект "Ряд Закладок" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | РядЗакладок**. Объект позволяет создавать многостраничные шаблоны, т.е. шаблоны на которых по нажатию закладок становятся видимыми соответствующие страницы с различными элементами управления. Сам объект не является контейнером для других объектов и лишь предоставляет механизм для визуализации ряда закладок и получения от него уведомлений при интерактивных действиях пользователя; переключение видимости различных элементов шаблона должно выполняться программистом в явном виде (путем кодирования).

Для отображения и изменения характеристик проигрывателя используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены страницы: "Общие", "Закладки", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для закладок.

Страница "Общие" позволяет задать *название объекта*, под которым он будет известен бланку и может использоваться в коде в качестве *переменной типа "РядЗакладок"*.

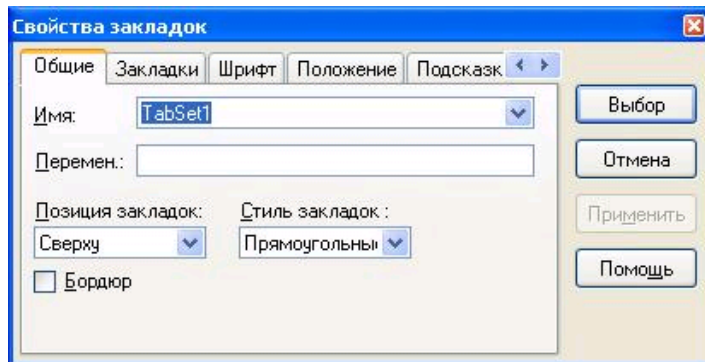


Рис. Страница "Общие" свойств ряда закладок.

Кроме того, данная страница содержит поле для указания имени целочисленной переменной, которая может быть связана с объектом. Значение этой переменной будет автоматически изменяться программой при изменении выделенной закладки, а также изменение этой переменной из программы позволит переключить закладки искусственным путем. Первая закладка соответствует значению переменной 1, вторая - 2, и так далее. Значение 0 означает, что ни одна закладка не является выделенной.

Флаг **Бордюр** указывает, следует ли по периметру объекта прорисовывать трехмерный бордюр.

Выпадающий список **Позиция закладок** определяет один из возможных способов ориентации закладок (т.е. способов визуальной индикации их размещения относительно контролируемых ими гипотетических страниц с элементами управления): сверху или снизу.

Выпадающий список **Стиль закладок** определяет способ отрисовки закладок: трапецевидные, прямоугольные, в виде кнопок.

Страница "Закладки" содержит редактируемый **Список**, где можно ввести произвольное количество строк - каждая строка задает новую закладку с соответствующим заголовком. Если в строке встречается символ "|", то всё, что расположено левее его, выводится в закладке (название), а всё, что расположено правее, - считается текстом всплывающей подсказки.

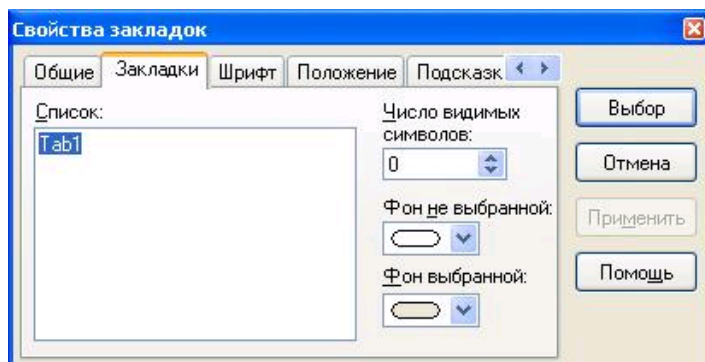


Рис. Страница "Закладки" свойств ряда закладок.

Поле **Число видимых символов** позволяет задать ограничение по ширине для закладки. Если здесь стоит ноль, ограничения нет, и ширина закладки выбирается таким образом, чтобы в ней уместился весь заголовок. Если значение больше нуля, программа будет обрезать более длинные заголовки.

Ниже расположены поля для выбора цветов фона выбранной и невыбранной закладок.

На странице "Подсказка" в дополнение к полю **Текст подсказки** расположен флаг **Брать текст из заголовка текущей закладки**. Если в поле **Текст подсказки** имеется какой-то текст, то всегда именно он выводится в качестве подсказки для всего элемента управления. Если же это поле пусто и включен флаг **Брать текст из заголовка текущей закладки**, то всплывающая подсказка может формироваться для тех закладок, заголовков которых не уместился по ширине в **Число видимых символов**, из полного заголовка закладки. Этот режим работает только в том случае, если соответствующие закладки не имеют явно заданной подсказки в своем заголовке после символа "|": если такой символ есть, то в качестве подсказки всегда используется строка после символа "|".

Объект "Список" может быть добавлен в шаблон бланка с помощью инструментальной кнопки < b>Вставить объект | Список.

Для отображения и изменения характеристик списка используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно также, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для списка.

Страница "Общие" позволяет задать *название объекта-списка*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Список"](#).

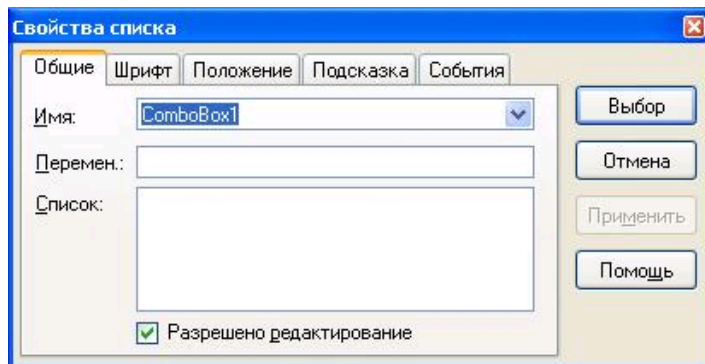


Рис. Страница "Общие" свойств списка.

Кроме того, данная страница содержит поле для перечисления элементов списка. Каждый элемент должен начинаться с новой строки.

В списке возможно использовать так называемую трансляцию. Фактически, *трансляция* - это механизм подмены отображаемых в списке значений на некоторые другие в момент их присвоения переменной, связанной со списком. Зачем это нужно, легко понять из следующего примера. Пусть в некотором бланке есть список, содержащий обозначения рода: "мужской" и "женский". Со списком связана переменная логического типа, причем мужской род обозначается значением FALSE, а женский - TRUE (почему именно так, а не наоборот, - не принципиально). При выборе из списка первой строки в переменную будет записано значение FALSE, а при выборе второй - TRUE.

Здесь требуется пояснить, что элементы списка нумеруются от 0 по возрастанию, а кроме того логическое значение FALSE эквивалентно 0 (нулю), а TRUE - 1 (единице). Сложив эти два факта вместе, мы и получим ответ на вопрос, почему первая строка связана с логическим значением FALSE, а вторая - TRUE.

Если бы значений было бы больше, то все они нумеровались бы по порядку, начиная с нуля (0, 1, 2 и т.д.), причем такой список должен быть связан уже не с логической, а целочисленной переменной (или переменными других типов, что определяется установками, описанными ниже в данном параграфе). Однако в любом случае, Студия неявно выполняет трансляцию, так как отображается в списке одно, а в переменную попадает другое. Прикладной программист может управлять трансляцией явно. Для этого в каждой строке-элементе списка нужно после текста, который будет отображаться на экране, указать через вертикальную черту транслируемое (внутреннее) значение. Например, если мы вдруг захотим поменять мужской и женский пол местами в вышеупомянутом списке, но при этом сохранить внутренние обозначения (мужской - FALSE, женский - TRUE), то строки можно записать следующим образом:

```
"женский| 1"  
"мужской| 0"
```

Также на странице "Общие" определяется, можно ли вручную вводить значения, отсутствующие в списке.

Если в объекте "Список" запрещен ручной ввод (флаг **Разрешено редактирование** на странице "Общие" не установлен), то с ним может быть связана логическая, целочисленная или строковая переменная.

Важно отметить, что *трансляция работает только в том случае, если редактирование в списке запрещено*.

Если ручное редактирование разрешено, то со списком может быть связана переменная любого типа по тем же правилам, что и с [объектом "Редактор"](#).

Объект "Флаг" может быть добавлен в шаблон бланка с помощью инструментальной кнопки **Вставить объект | Флаг**.

Для отображения и изменения характеристик флага используется унифицированное диалоговое окно, описанное в разделе [Окно настройки свойств](#). В левой его части расположены пять страниц: "Общие", "Шрифт", "Положение", "Подсказка" и "События".

Страницы "Шрифт", "Положение", "Подсказка" и "События" выглядят и действуют точно так же, как и для [всех объектов](#). При этом следует иметь в виду, что на странице "События" выводится перечень событий, специфических именно для флага.

Страница "Общие" позволяет задать *название флага*, под которым он будет известен бланку и может использоваться в коде в качестве [переменной типа "Флаг"](#), а также надпись, выводимую справа от флага.

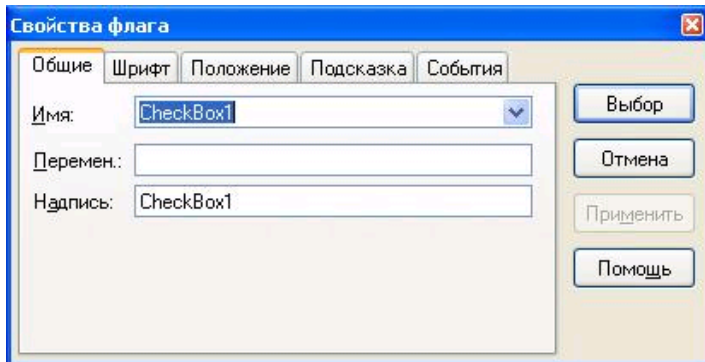


Рис. Страница "Общие" свойств флага.

Кроме того объект "Флаг" можно связать с переменной бланка, имя которой указывается в одноименном поле. Связывание переменной с объектом позволяет синхронно изменять значение переменной вместе с состоянием объекта. Так, объект "Флаг" может быть связан с переменной одного из типов: логического, целое, перечисление ([type](#)). При этом каждый раз при установке флага в нее будет записываться значение Истина (1 или элемент перечисления, соответствующий 1), а при его снятии - Ложь (0 или элемент перечисления, соответствующий 0). Следует иметь в виду, что для правильной работы в привязке к флагу перечисление должно иметь элементы со значениями 0 и 1, например:

```
type VSex = (varMale = 0, varFemale);
```

Для отображения и настройки свойств фрейма шаблона в окне настройки свойств используются шесть страниц: "[Общие](#)", "[Размер](#)", "[Прочие](#)", "[Субфреймы](#)", "[Подсказка](#)", и "[События](#)".

Страница "Общие"

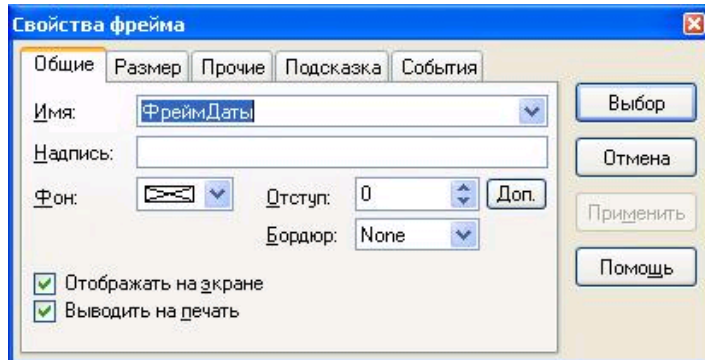


Рис. Страница "Общие" свойств фрейма.

Поле **Имя**

Поле предназначено для задания имени фрейма, по которому он будет идентифицироваться программой. Введенное здесь имя фрейма используется при обращении к данному объекту в исходных кодах программы, а также оно отображается в дереве фреймов текущего шаблона в [структуре документа](#).

Поле **Надпись**

Вводится произвольный текст, который выводится на закладке.

Поле **Фон**

Позволяет назначить цвет фона фрейма, позволяющий повысить наглядность бланка и отличить один фрейм от другого.

Поле **Отступ** и кнопка **Доп**

В поле указывается *одинаковая для всех сторон* величина отступа содержимого от края фрейма. Величина отступа измеряется в **пикселах**. Кнопка **Доп** открывает окно, в котором можно настроить отступы для каждой из сторон фрейма по отдельности. По умолчанию в полях проставляется значение, заданное в поле **Отступ**.

Поле **Бордюр**

При выборе значения NoneBevel бордюр вокруг фрейма отсутствует, в остальных случаях прорисовывается бордюр в соответствии с выбранным стилем: по умолчанию (DefaultBevel), тонкий (SingleBevel), статический (StaticBevel), внутренний (ClientBevel) или оконный (WindowBevel) бордюр.

Флаг **Отображать на экране**

Если флаг установлен, текущий фрейм будет отображаться на экране, при снятом флаге фрейм не виден.

Флаг **Выводить на печать**

При установке флага содержимое фрейма будет выводиться на печать, иначе - нет.

Страница "Размер"

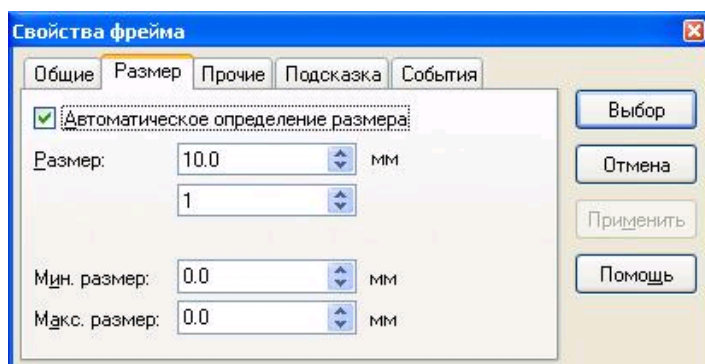


Рис. Страница "Размер" свойств фрейма.

Флаг **Автоматическое определение размера**

Если флаг установлен, то разрешается вычисление размеров фрейма в зависимости от его содержимого, при снятом - запрещается.

Поле **Размер**

В верхнем поле можно явно указать размер фрейма в мм. Если поле не заполнено, размер фрейма рассчитывается, исходя из той доли, которая указывается в нижнем поле.

Поля **Мин. размер** и **Макс. размер**

В данные поля вводится соответственно минимальный и максимальный размер фрейма, которые ограничивают значение, заданное в поле **Размер** снизу и сверху.

Страница "Прочие"

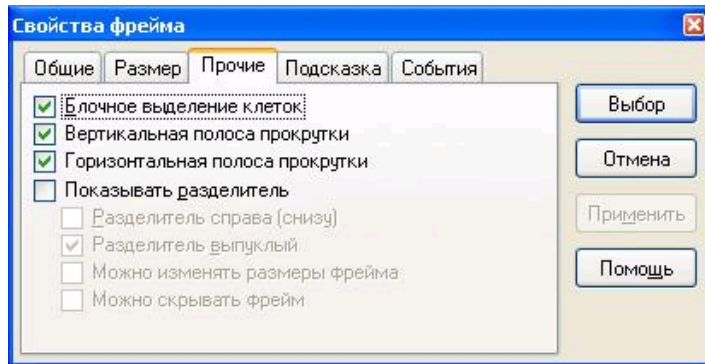


Рис. Страница "Прочие" свойств фрейма.

Флаг **Блочное выделение клеток**

При установленном флаге разрешается выделять на экране группу клеток из одной секции, нескольких секций (квадратный блок), нескольких строк (строковый), нескольких столбцов (столбцовый блок). По умолчанию флаг установлен.

Флаги **Вертикальная полоса прокрутки**, **Горизонтальная полоса прокрутки**

Данные флаги позволяют отображать как обе полосы вместе (установлены оба флага), так и отдельно, одну из них (установлен один флаг). При снятых флагах не видна ни вертикальная, ни горизонтальная полоса прокрутки.

Флаг **Показывать разделитель**

При установке данного флага разделитель отображается на шаблоне, при снятом флаге становятся недоступными все нижележащие флаги.

Флаги **Разделитель справа (снизу)**, **Разделитель выпуклый**

Если флаг **Разделитель справа (снизу)** установлен, то разделитель отображается справа (при вертикальной ориентации) или снизу (при горизонтальной ориентации). При снятом флаге разделитель отображается слева/вверху. Причем, если флаг **Разделитель выпуклый** выключен, то разделитель является плоским, иначе - выпуклым.

Флаг **Можно изменять размеры фрейма**

Если флаг установлен, то разрешается изменять размеры фрейма с помощью мыши, перемещая разделитель, иначе - не разрешается.

Флаг **Можно скрывать фрейм**

При установленном флаге на разделителе этого фрейма появляется, так называемая область закрытия, щелкнув на которую можно сделать фрейм невидимым.

Страница "Субфреймы"

Страница появляется в диалоге при наличии на шаблоне двух и более фреймов.

Поле **Ориентация**

В поле задается один из возможных способов размещения фреймов на шаблоне. Фреймы на шаблоне могут располагаться друг под другом (горизонтальная ориентация), занимать вертикальные полосы, или иметь смешанную ориентацию, накладываться друг на друга (ориентация с закладками). В этом режиме перемещение по фреймам происходит как в многостраничном диалоге по закладкам. Имя закладки указывается [на странице "Общие"](#) этого диалога в поле **Надпись**. Если поле не заполнено, то в качестве заголовка закладки берется значение поля **Имя**.

Если выбрана ориентация "С закладками", то на странице появляется перечисленные ниже элементы

управления.

Флаг **Показывать закладки**

При включенном флаге закладки отображаются на шаблоне, при снятом флаге - не отображаются, и все элементы, задающие различные атрибуты закладок, становятся недоступными.

Поле **Позиция закладок**

Указывается местоположение закладок на фрейме: сверху, слева, справа, снизу.

Поле **Стиль закладок**

Выбирается внешний вид отображения закладок: в форме трапеций, прямоугольников или кнопок.

Флаг **Авто-размер**

Флаг позволяет включать (установлен) и отключать (снят) режим масштабирования закладок фреймов. По умолчанию флаг выключен.

Флаг **Как TabControl**

Если флаг установлен, то появляется рамка, обрамляющая весь контур фрейма, как в обычных диалогах, при снятом флаге - рамка отсутствует.

Страница "Подсказка"

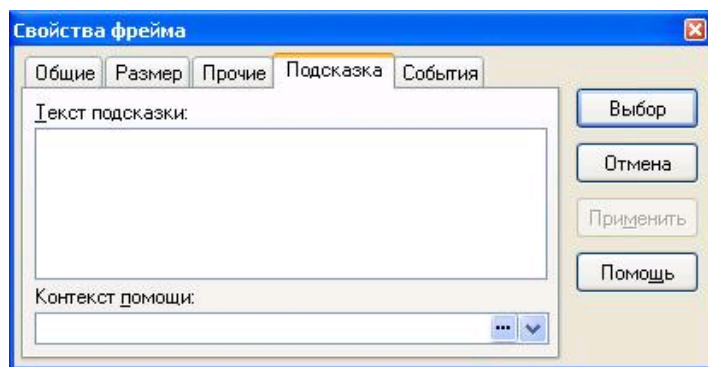


Рис. Страница "Подсказка" свойств фрейма.

Поле **Текст подсказки**

В поле вводится текст для подсказки, который будет отображаться, если курсор установлен на закладке фрейма и выполнены следующие условия: [на странице "Субфреймы"](#) выбран способ ориентации фреймов - с закладками и включен флаг **Показывать закладки**.

Поле **Контекст помощи**

В поле указывается путь к файлу с текстом помощи, поясняющий назначение данного фрейма. Этот текст будет появляться, если нажата клавиша F1 и фокус установлен на каком-либо из объектов заданного фрейма.

Страница "События"

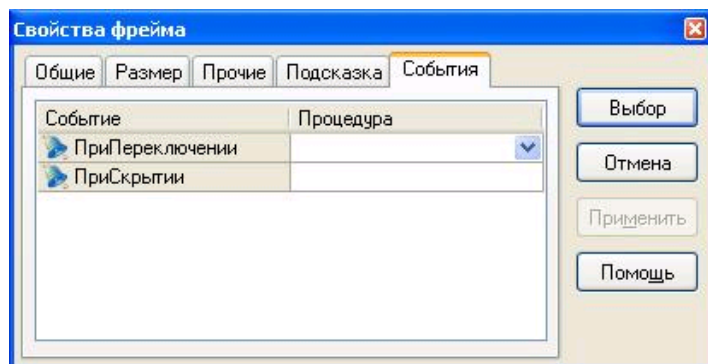


Рис. Страница "События" свойств фрейма.

Страница содержит список [событий](#), предусмотренных для объекта фрейм. В колонке **Процедура** для нужного события указывается имя обработчика события, используемого во фрейме.

Настройка шаблона выполняется в диалоге "Свойства шаблона", который содержит четыре страницы: "[Общие](#)", "[Окно](#)", "[Печать](#)" и "[События](#)". Вызов диалога осуществляется командой **Свойства шаблона** всплывающего меню, доступной из текущего окна шаблона в дизайн-режиме.

Страница "Общие"

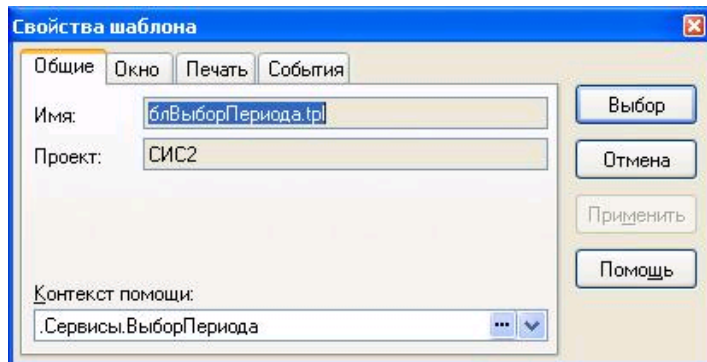


Рис. Страница "Общие" свойств шаблона.

Поля **Имя** и **Проект**

В этих полях отображается имя двоичного файла с шаблоном (TPL-файла, формируемого визуальным редактором шаблонов) и имя проекта, в котором был разработан этот шаблон.

Поле **Контекст помощи**

В поле указывается путь к файлу с темой справки. Для ее вызова следует открыть шаблон в дизайн-режиме и нажать клавишу **F1**. При отсутствии сведений (темы справки) по работе с шаблоном поле не заполнено.

Страница "Окно"

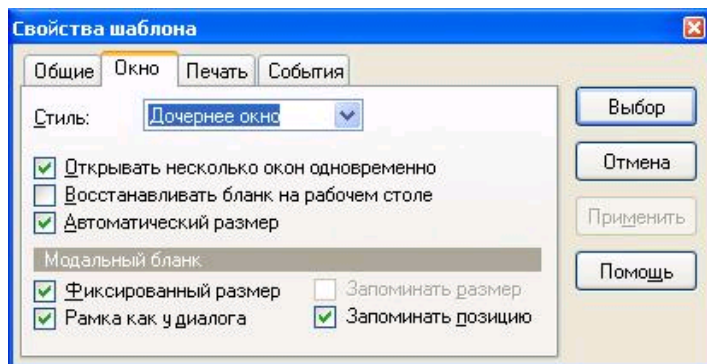


Рис. Страница "Окно" свойств шаблона.

Страница "Окно" предназначена для установки свойств, связанных с логикой поведения окна бланка с данным шаблоном.

Поле **Стиль**

Выпадающий список данного поля содержит набор стилей окна, применяемых программой для идентификации видов окон. Наличие стилей позволяет открыть шаблон в выбранном режиме, например, как дочернее окно, модальное окно, плавающее окно или встраиваемое окно. Если выбран способ открытия шаблона - Модальное окно, он открывается модально.

Использование стиля встраиваемых окон разрешает прижимать такие окна к любому краю главного окна и вставлять их друг в друга наравне с окнами инструментария. По существу встраиваемое окно - это плавающее окно, которое может быть прижатым к каким-либо границам главного окна. Плавающее окно может только перемещаться (плавать) поверх остальных окон, но не может быть прижатым ни к какому краю.

Флаг **Открывать несколько окон одновременно**

Если флаг включен, то программа разрешит одновременное заполнение двух бланков на основе текущего шаблона. В противном случае при повторном вызове бланка будет активизироваться уже открытый экземпляр окна.

Флаг **Восстанавливать бланк на рабочем столе**

Флаг влияет на то, как программа обрабатывает открытый бланк на стадии запуска и закрытия сессии. В конце сессии программа запоминает все бланки, которые имеют данный флаг во включенном положении и были в тот момент открыты, а при последующем запуске - автоматически восстанавливает эти бланки на рабочем столе (внутри главного окна программы).

Флаг **Автоматический размер**

При установке флага происходит автоматическое изменение размеров окна шаблона при изменении его содержимого, в частности этот режим решает проблемы работы с крупными шрифтами. По умолчанию флаг выключен, такой режим запрещает автоматически изменять размер шаблона. Рекомендуется включать флаг у всех бланков, работающих как модальные диалоги.

Группа флагов **Модальный бланк**

Эти флаги влияют на работу бланка только, когда он открыт в модальном режиме.

Флаг **Фиксированный размер**

Если флаг включен, то пользователь не имеет возможности изменить размер окна.

Флаг **Рамка как у диалога**

Установка флага отображает модальное окно в виде окна диалога или же стандартного окна Windows.

Флаги **Запоминать размер** и **Запоминать позицию**

Установка этих флагов предписывают программе запоминать размер и позицию бланка каждый раз, когда он закрывается, и восстанавливать эти значения при последующем его открытии.

Страница "Печать"

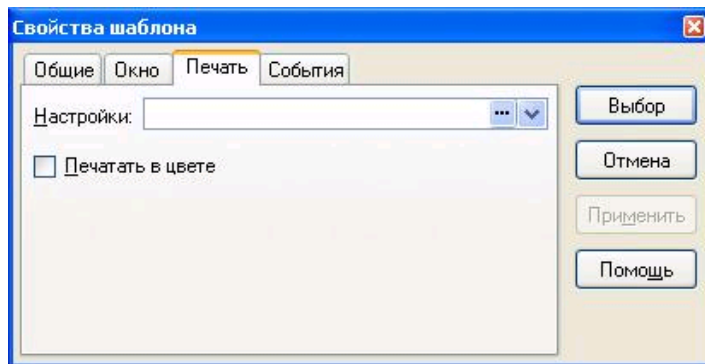


Рис. Страница "Печать" свойств шаблона.

Поле **Настройки**

Программа разрешает создавать настроечные файлы (схемы), в которых будут записаны параметры печати шаблона (размер и ориентация страницы, колонтитулы и т.д.), именно это имя настроечного файла и надо задать в данном поле. Параметры печати шаблона (размер и ориентация страницы, колонтитулы и т.д.) определяются в диалоге ["Настройка печати"](#). Если несколько шаблонов имеют один и тот же настроечный файл, то использующие их бланки будут выводиться на печать с одними и теми же параметрами.

Флаг **Печать в цвете**

По умолчанию флаг снят, поэтому шаблон распечатывается в черно-белом цвете. Установка флага позволяет распечатать его в цветном виде, но *при наличии цветного принтера*.

Страница "События"

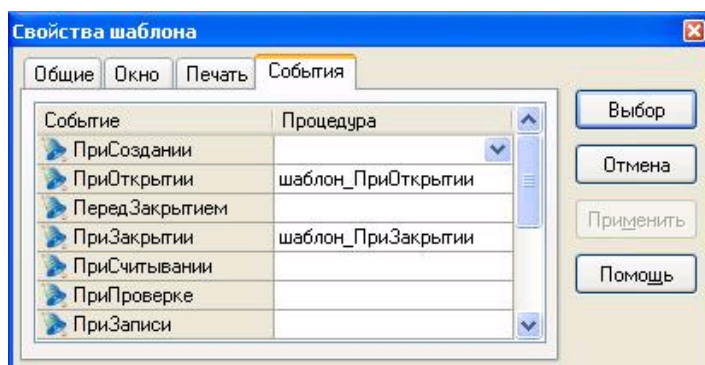


Рис. Страница "События" свойств шаблона.

Страница содержит список [событий](#), предусмотренных для бланка, где прикладной программист может задать имена обработчиков событий.

Для изменения параметров клетки, группы клеток, объекта или всего шаблона используется унифицированное немодальное диалоговое *окно настройки свойств*, состоящее из нескольких страниц. Оно открывается по команде **Свойства клеток** для клеток и их групп, **Свойства шаблона** - для шаблона и **Свойства** - для интерфейсных объектов. Все эти команды доступны из контекстного меню, вызываемого нажатием правой кнопки мыши над соответствующим элементом.


В правой части окна свойств расположены следующие кнопки:

- **Выбор** - применение произведенных в данном диалоге изменений к текущей клетке, группе клеток, объекту или шаблону и закрытие окна;
- **Отмена** - закрытие окна без применения изменений;
- **Применить** - применение изменений без закрытия окна;
- **Помощь** - вызов Справочной системы с темой, зависящей от назначения объекта.

Поскольку данный диалог является немодальным, он допускает перемещение курсора по шаблону. В процессе перемещения в нем отображаются свойства текущего объекта.

В общем случае [диалоги](#), свойств объектов содержит следующие страницы: ["Общие"](#), ["Шрифт"](#), ["Положение"](#), ["Подсказка"](#), ["События"](#). Страница "Формат" в этой теме не описывается, т.е. имеет специфические элементы управления.

Страница "Общие"

Страница "Общие" содержит элементы управления, специфические для каждого класса объектов, поэтому работа с ней описывается далее в соответствующих разделах. Для всех объектов на странице "Общие" всегда имеется поле **Имя**, где вводится идентификатор объекта. Все объекты шаблона перечисляются в выпадающем списке, который открывается кнопкой . Выбор нового объекта из списка приводит к переключению на него (он становится выделенным), и к смене диалога свойств, в котором отображаются свойства выбранного объекта.

Для удобства программиста Студия позволяет автоматически формировать в исходном коде бланка переменные объектных типов, связанные с элементами шаблона. Для этого достаточно ввести требуемое имя элемента в поле **Имя** и нажать кнопку **Применить**. Если такого объекта еще не было описано в исходном коде, программа добавит его описание в раздел [InObject](#) и [Private](#) бланка. Если новое имя было введено не в результате создания нового объекта, а переименования старого, Студия также автоматически изменит имя соответствующей переменной в коде бланка. По двойному щелчку мыши в поле **Имя** осуществляется переход в исходный текст бланка в то место, где описана связанная с объектом переменная.

Остальные четыре страницы одинаковы для всех объектов.

Страница "Шрифт"

Страница "Шрифт" служит для установки *параметров шрифта*, которым на объекте будет выводиться текст. К ним относятся название шрифта, его размер, начертание (жирное, курсив), а цвета текста (поле **Текст**) и фона (поле **Фон**) для текущего объекта. В поле **Образец** можно просмотреть, как будет выглядеть текст на объекте с учетом всех сделанных настроек. Настройка полей и элементов управления на этой странице выполняется так же, как и в диалоге [Выбор шрифта](#).

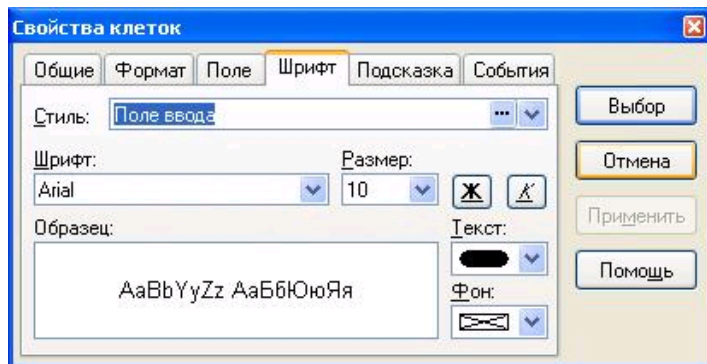


Рис. Страница "Шрифт".

На странице "Положение"

На странице "Положение" задается расположение объекта на шаблоне и его размеры (все величины указываются в миллиметрах), а также определяется, следует ли отображать объект при печати данного

бланка на принтере. На этой же странице также имеется флаг **Переходить табуляцией**, доступный для большинства типов объектов. Исключение составляют лишь **Надпись** и **Рамка**. Если данный флаг включен, объект попадает во внутренний общий список объектов шаблона, по которому можно переключать фокус ввода с одного объекта на другой, последовательно нажимая **Tab** (прямой порядок) или **Shift+Tab** (обратный порядок). Объект, имеющий фокус ввода, выделяется на бланке пунктирным прямоугольником и получает клавиатурные команды пользователя. Кроме объектов, помеченных данным флагом, в список табулируемых элементов шаблона включаются и все клетки с редактируемыми данными и клетки формата, предполагающего интерактивное взаимодействие с пользователем (например, поле формата "статический текст" в виде кнопки).

Положение и размеры объектов можно менять с помощью мыши непосредственно на шаблоне. Перемещается объект методом "перетащи и отпусти", то есть достаточно навести курсор на объект (курсор мыши при этом меняет свою форму на "указующий перст"), нажать левую кнопку и, не отпуская ее, перетащить объект на новое место. Изменение размеров выполняется во многом аналогично, за исключением того, что курсор требуется наводить на границу объекта (курсор при этом меняет форму на двунаправленную стрелку).

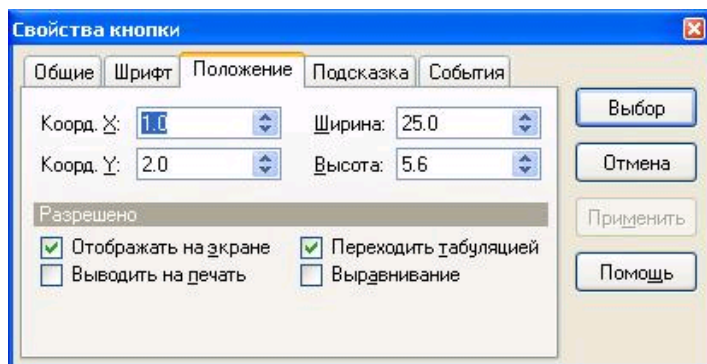


Рис. Страница "Положение".

Страница "Подсказка"

Страница "Подсказка" дает возможность ввести произвольный текст, поясняющий назначение объекта. В поле **Текст подсказки** вводится текст для подсказки, который будет отображаться, если курсор установлен на объекте.

В поле **Контекст помощи** указывается путь к файлу с темой помощи, поясняющий назначение данного объекта. Этот тема будет показываться в Справочной системе во всплывающем окне, если нажать клавишу **F1** и установить фокус на объекте.

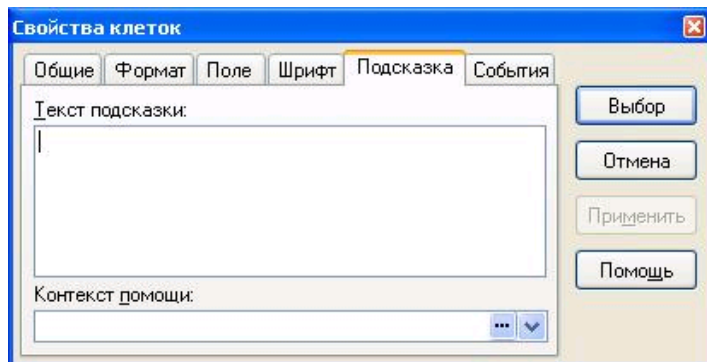


Рис. Страница "Подсказка".

Страница "События"

Страница "События" содержит список [событий](#), предусмотренных для объекта данного типа. Разработчик имеет возможность назначить для событий методы-обработчики, введя их имена в соответствующие клетки в колонке **Процедура**.

Список событий, предусмотренных для текущего объекта, и требования к их обработчикам приведены в разделе, посвященном объекту соответствующего типа. Система позволяет автоматически сгенерировать заготовку обработчика требуемого синтаксиса по двойному щелчку мыши в соответствующей клетке или выбрать один из имеющихся методов с подходящим прототипом.

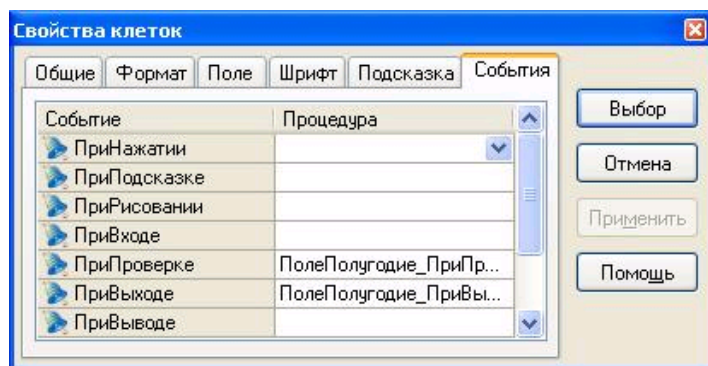


Рис. Страница "События".

Разделение шаблона документа на секции позволяет создавать документы, не имеющие строгой табличной формы или состоящие из повторяющихся частей, различных между собой.

Для повышения гибкости механизма шаблонов в визуальный редактор были добавлены специальные возможности размещения информации. К ним относятся операции *разделения* и *объединения секций*, а также возможность *объединения нескольких клеток внутри одной секции*.

[Разделение и объединение секций](#)

[Объединение нескольких клеток секции](#)

[Вставка секции](#)

[Настройка свойств секции](#)

[Выделение клеток секции. Удаление клеток](#)

[Добавление строк в секцию](#)

[Настройка свойств строк](#)

[Добавление столбцов в секцию](#)

[Настройка свойств столбцов](#)

Вставка новой секции происходит с помощью одноименной команды, доступной в стандартной настройке Студии из всплывающего меню в дизайн-режиме бланка. При этом на экране появляется диалог "Вставка секций" (см. рис. Вставка секций).

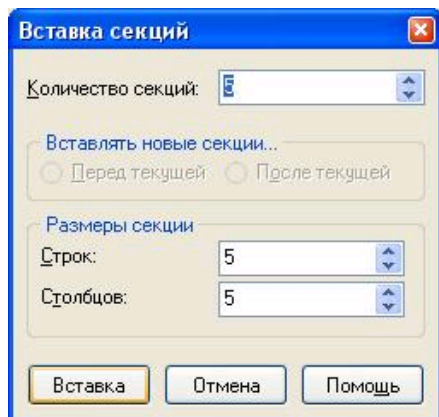


Рис. Вставка секций.

В диалоге следует указать:

- количество вводимых секций (поле **Количество секций**);
- расположение добавляемых в шаблон секций относительно текущей секции, в которой находится курсор- подсвеченная клетка, или секции, отмеченной на горизонтальной и вертикальной линейках белым полем. Размещение секции по высоте шаблона (до или после текущей секции) определяется выбранной позицией переключателя **Вставлять новые секции**. Секция всегда выровнена по левому краю шаблона, поэтому регулировать размещение вставляемой секции по ширине (отступ вправо или влево относительно текущей секции) можно последующим добавлением пустых столбцов соответствующей ширины во вставляемую секцию.
- количество строк и столбцов в секции|секциях (в группе полей **Размеры секции**).

Размеры строк и столбцов задаются (уже после вставки секции) в диалогах "Свойства строк" и "Свойства столбцов" или мышью с использованием линеек.

Для повторяющихся секций дополнительно указываются их специфические свойства в диалоге "Свойства секций".

Впоследствии количество столбцов и строк секции доступно для изменения с помощью команд **Вставка строк** (сразу несколько), **Вставить строку** (одну), **Вставка столбцов** (сразу несколько), **Вставить столбец** (один), **Удалить строку**, **Удалить столбец**, а также диалога настройки свойств секции.

Вставка строк в секцию

После [выделения строкового блока](#) во всплывающем меню дизайн-режима, которое вызывается по нажатию правой кнопки мыши, появляются команды [Вставка строк](#) и [Свойства строк](#).

Вставка строк производится с помощью диалога "Вставка строк".

Как и в диалоге [вставки новой секции](#), при добавлении строк указываются число вставляемых в секцию строк и *позиция вставки* (до или после текущей строки).

В визуальном редакторе шаблонов бланков существует несколько способов выделения клеток:

- выделение строк (строковый блок);
- *выделение столбцов (столбцовый блок);*
- *прямоугольный блок;*
- *выделение всех клеток шаблона.*

Удобнее всего производить выделение клеток с помощью мыши. В этом случае для выделения строк достаточно нажать левую кнопку мыши на вертикальной линейке (указатель принимает форму горизонтальной стрелки) и, не отпуская ее, перемещать указатель вверх или вниз; для выделения столбцов следует проделать аналогичные манипуляции с горизонтальной линейкой (курсор в этом случае выглядит как вертикальная стрелка); потоковый и прямоугольный блоки выделяются простым перемещением указателя мыши по клеткам с зажатой левой кнопкой; для выделения всех клеток шаблона требуется произвести двойной щелчок мышью в левом верхнем углу шаблона на пересечении линеек.

Все блоки, за исключением строкового блока, ограничиваются пределами одной секции.

Для удаления содержимого выделенного блока используется команда **Удаление**. Если в качестве блока помечена строка, столбец, секция или группа секций, то они будут удалены из шаблона.

Добавление столбцов в секцию

После [выделения строкового блока](#) во всплывающем меню дизайн-режима, которое вызывается по нажатию правой кнопки мыши, появляются команды [Вставка столбцов](#) и [Свойства столбцов](#).

Вставка столбцов производится с помощью диалогового окна "Вставка столбцов".

В диалоге при добавлении столбцов указываются *число* вставляемых столбцов и *позиция вставки* (перед или после текущего столбца).

После добавления секции в шаблон можно настроить ее параметры, воспользовавшись командой [Свойства секций](#). В результате открывается диалог "Свойства секции", в котором отображаются параметры текущей секции.

На первой странице данного диалога - "Общие" - в соответствующее поле вводится имя секции, которое должно быть уникальным в шаблоне.

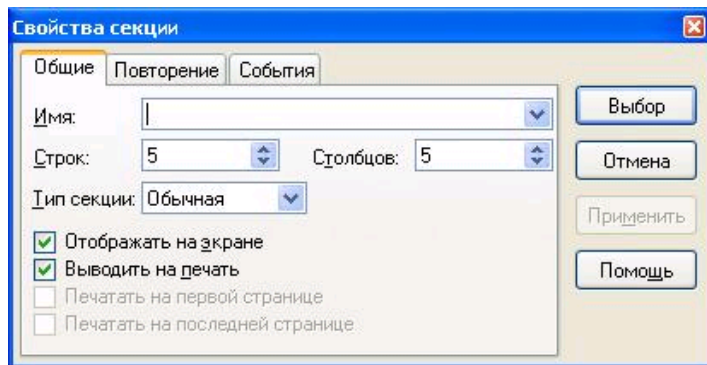


Рис. Страница "Общие" свойств секции.

На этой же странице задается количество строк и столбцов в кадре, а также с помощью соответствующих флагов указывается, следует ли выводить секцию на экран и на печать. Таким образом, возможно создание скрытых секций и секций, видимых лишь на экране, но не выводимых на печать (или наоборот).

На второй странице "Повторение" с помощью соответствующего флага следует определить, является ли секция повторяющейся. Если этот флаг установлен, то дополнительно следует разрешить или запретить управление секцией вручную, для чего предназначена группа управляющих элементов **Разрешено**. В ней расположено 3 флага: **Добавлять строки**, **Удалять строки** и **Перемещать строки**. Если соответствующий флаг включен, то пользователь может в режиме заполнения бланка выполнять операции добавления, удаления и перемещения кадров. В противном случае соответствующая операция может быть выполнена лишь программным способом. Если все флаги сброшены, секция недоступна для редактирования.

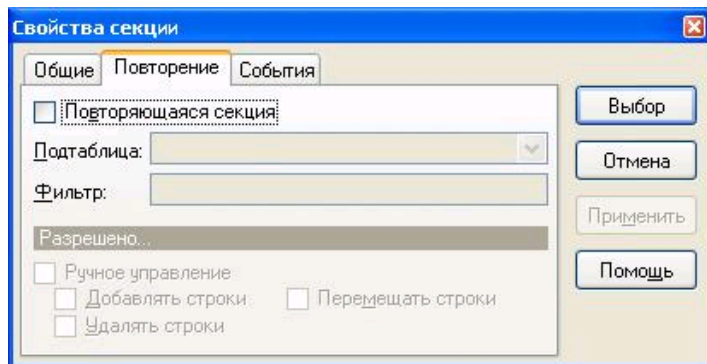


Рис. Страница "Повторение" свойств секции.

Также для повторяющихся секций бланка-редактора можно указать имя подтаблицы редактируемого документа, с которой связана данная секция. Например, если в электронном документе, описывающем накладную, имеется таблица с перечислением позиций, то ее имя следует указать в поле ввода **Подтаблица**. После этого повторяющаяся секция будет автоматически заполняться перечнем товаров из записи в базе данных при открытии конкретного документа на редактирование.

Поле **Фильтр** позволяет задать строковое выражение логического типа для отбора записей из подтаблицы, которые следует отображать в секции. В секции выводятся только те записи подтаблицы, для которых истинно выражение фильтра.

На третьей странице при необходимости задаются обработчики событий, которые могут происходить в секции.

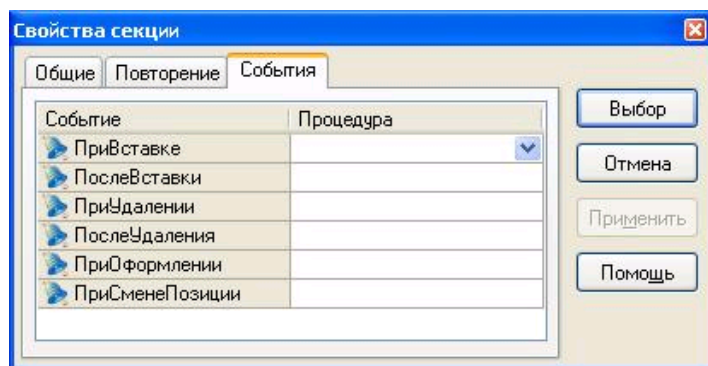


Рис. Страница "События" свойств секции.

Настройка свойств столбцов

Для настройки свойств столбца используется диалог "Свойства столбца", открываемый по команде [Свойства столбцов](#). Команда доступна из контекстного меню, которое открывается по нажатию правой кнопки мыши на верхней (горизонтальной) [линейке](#).

Окно настройки свойств столбца позволяет установить ширину всех столбцов в выделенном блоке, а также определить, следует ли отображать выделенные столбцы на экране и выводить на печать.

Настройка свойств строк

Для настройки свойств строк используется диалог "Свойства строки", вызываемый по команде [Свойства строк](#). Команда доступна из контекстного меню, которое открывается по нажатию правой кнопки мыши на левой (вертикальной) [линейке](#).

Окно настройки свойств строки позволяет установить способ определения высоты строк, входящих в выделенный блок. В выпадающем списке **Определение высоты** возможно выбрать одну из трех альтернатив:

- "Автоматически" - высота строки будет определяться программой в зависимости от параметров используемых в ней шрифтов;
- "Не менее чем..." - высота строки не может быть меньше значения, заданного в поле **Высота (мм)**;
- "Точно" - высота строки должна быть равной значению, заданному в поле **Высота (мм)**.

Кроме того, можно установить флаги **Отображать на экране**, **Выводить на печать** и **Печатать с новой страницы**. Установка последнего флага позволяет при печати бланка переводить страницу перед данной строкой.

Объединение нескольких клеток секции

Даже если документ нельзя представить в виде регулярной таблицы, вовсе не обязательно использовать различные секции. Часто достаточно объединить несколько лежащих рядом по горизонтали или вертикали клеток одной секции в единую клетку. При этом все свойства, установленные для левой верхней клетки такого объединения, будут распространяться на всю новую область.

Для объединения нескольких соседних клеток необходимо их сначала выделить, а затем выполнить команду [Слияние клеток](#) (например, с помощью кнопки на панели инструментов). Разъединение объединенных клеток выполняется с помощью той же, повторно примененной, команды.

Объединение клеток уменьшает число секций в документе, позволяя при этом размещать данные в шаблоне достаточно произвольно. Поэтому практически единственной причиной, по которой следует создавать различные секции, является потребность в использовании [повторяющихся секций](#).

Секция может содержать один или несколько кадров, т.е. наборов смежных строк и столбцов, содержащих логически связанные клетки. Секции, которые могут содержать несколько кадров, называются повторяющимися и позволяют формировать документы переменной длины. Все кадры одной секции имеют одинаковую структуру - количество строк и столбцов, а также настройки их клеток.

В дизайн-режиме в любой секции содержится по одному кадру.

Например, в бланке счета-фактуры секции с заголовком счета-фактуры и итоговыми суммами должны существовать в единственном экземпляре, в то время как в состав перечня товаров может входить несколько строк, причем их количество заранее неизвестно. В этом случае можно определить секцию с перечнем товаров как повторяющуюся, а ее кадр оформить в виде таблицы из одной строки, содержащей реквизиты "Наименование товара", "Цена", "Количество" и т.д. В соответствующие клетки кадра целесообразно вывести переменные бланка. Тогда при переходе рассматриваемого бланка в режим заполнения можно будет добавлять в нее новые кадры (в нашем случае - строки). Таким образом, счет-фактура может содержать переменное число позиций.

Если количество кадров в секции равно нулю, секция на экран не выводится. Это свойство повторяющихся секций можно использовать для создания документов переменного состава.

Например, в одном и том же бланке требуется выводить либо набор реквизитов, характеризующих физическое лицо, либо набор реквизитов организации. Их можно оформить в виде двух повторяющихся секций таким образом, чтобы количество рамок в одной из них было равно 0, а во второй - 1, и наоборот. То есть, манипулируя количеством кадров программно (что достигается установкой свойства **Количество** объекта [СекцияШаблона](#), см. *Справочник по языку ТБ.Скрипт*), можно показывать в бланке либо один, либо другой набор реквизитов.

Повторяющиеся секции очень удобны для работы с [переменными-массивами](#), т.е. с переменными бланков, которые содержат в себе несколько пронумерованных значений одного типа. Для связи с такой переменной в поле бланка достаточно указать ее имя после символа "#" в клетке, входящей в повторяющуюся секцию. Тогда при переходе бланка в режим заполнения полей каждое повторение кадра будет содержать значение из соответствующего элемента переменной массива: на первом повторении - из первого, на втором - из второго и т.д.

Если бланк является [бланком-редактором](#) некоторой записи (картотеки), то в нем аналогично организуется и работа с информацией, содержащейся в многозначных полях картотек (в массивах и периодических полях записей). Более подробная информация об этом содержится в главе, посвященной [подсистеме картотек](#).

Для программного управления секциями, в том числе - повторяющимися, в иерархии классов Студии реализовано несколько классов - [СекцияШаблона](#), [СтолбецШаблона](#), [СтрокаШаблона](#) - с богатым набором свойств и методов, например, **ВставитьКадр**, **УдалитьКадр**, **УпорядочитьПо** служат соответственно для добавления или удаления кадра, а также сортировки кадров секции по столбцу. Полная информация об этих и других классах (включая процедуры и функции работы с повторяющимися секциями) приведена в [Справочнике по объектному языку ТБ.Скрипт](#).

Повторяющаяся секция может иметь ручное управление. В этом случае минимальное число входящих в нее кадров равно единице.

Следует иметь в виду, что в повторяющейся секции свойства клеток и строк из разных кадров не могут отличаться. Иными словами, изменение свойств какой-либо клетки или строки любого кадра также сказывается на всех остальных кадрах. Все кадры имеют одинаковые настройки. При этом допускается делать различные настройки свойств для разных клеток и разных строк внутри одного кадра.

Часто в процессе редактирования шаблона требуется разделить секцию, изначально казавшуюся однородной, на части, имеющие различное число столбцов. Такая возможность предоставляется визуальным редактором бланков Студии путем использования команд **Разрезать секцию** и **Склеить секции**. Обе команды по умолчанию отсутствуют в меню Студии (напомним, что пользователь может [настроить меню](#) по-своему), однако на вкладке "Шаблон" панели инструментов есть соответствующие кнопки.

С помощью команды **Разрезать секцию** одна горизонтальная секция разделяется на две, при этом строка, на которой в момент выполнения программы стоял курсор, становится последней строкой верхней секции. После разделения формат вновь получившихся секций можно изменять независимо друг от друга.

Обратную операцию - соединение двух секций в одну - выполняет команда **Склеить секции**. Поскольку у объединяемых секций может быть различное число столбцов и разный размер, то при объединении соблюдаются следующие правила:

- число столбцов во вновь образовавшейся секции равно максимальному числу столбцов в объединяемых секциях;
- ширина столбцов определяется по верхней секции.

В наиболее общем случае шаблон бланка представляет собой совокупность следующих друг за другом по вертикали двумерных таблиц, называемых *секциями*. Каждая из них имеет определенное число строк и столбцов, которое может не совпадать с аналогичными параметрами других секций этого же шаблона, а также другие настройки, определяющие способ отображения информации в бланке. Использование нескольких секций в одном шаблоне позволяет создавать электронные варианты документов, имеющих сложную структуру расположения реквизитов.

Например, бланк счета-фактуры разделен на три логические части: заголовок, содержащий номер счета и реквизиты контрагентов, перечень товаров, отпускаемых по счету, а также заключительную часть с итогами и подписями представителей сторон. Каждая из частей имеет собственную структуру реквизитов, отличающуюся от остальных. Поэтому шаблон бланка "Счет-фактура" следует построить из трех секций, каждая из которых реализует свою часть документа.

Клетки, входящие в секцию, образуют так называемый *кадр* (*frame*). По сути дела, кадр - это горизонтальный фрагмент секции, состоящий из одной или нескольких логически взаимосвязанных соседних строк. Строки кадра либо присутствуют в секции все вместе, либо нет ни одной из них, то есть кадр, даже будучи составлен из нескольких строк, обладает свойством целостности (или неделимости). В простейшем, наиболее распространенном случае в кадре всего одна строка.

Предназначение кадров станет понятным, как только мы скажем, что они могут повторяться в секции. Иными словами, секция может содержать не только один, но и несколько однотипных кадров. Например, если в секции задан кадр из трех строк, то в секции может быть 0 строк (ни одного кадра), 3 строки (1 кадр), 6 строк (2 кадра) и так далее. Если в каждой строке кадра выводится определенный набор переменных-массивов (а именно так, как правило, и используются секции), то каждый кадр будет содержать элементы этих массивов с нарастающим индексом.

Секции, содержащие несколько кадров, называются повторяющимися. С помощью соответствующих методов и свойств класса **СекцияШаблона** (см. *Справочник по языку ТБ.Скрипт*) прикладной программист может изменять характеристики секций в процессе выполнения программы, например, определять число кадров в секции, добавлять и удалять кадры.

В то же время, существует возможность запретить повторение кадров, и такая секция называется *неповторяющейся*.

Использование повторяющихся секций при построении шаблонов бланков рассматривается в разделе [Повторяющиеся секции](#).

На стадии инициализации бланка в режиме выполнения каждая секция имеет 0 кадров (и 0 строк), несмотря на то, что в дизайн-режиме всегда отображается 1 кадр (или 1 строка). Затем секция может быть тем или иным образом заполнена (увеличение числа кадров или строк, программный или интерактивный ввод данных).

При переключении бланка в дизайн-режим (например, по нажатию *Ctrl+D* в режиме отладки бланка или в результате попытки его отредактировать в режиме проектирования) открывается окно редактора шаблонов, в котором бланк отображается несколько иначе, чем во время исполнения проекта. Практически все видимые изменения служат для облегчения процесса разработки шаблона и носят функциональный характер. Например, в дизайн-режиме все клетки шаблона (вне зависимости от того, как они отформатированы) обводятся по периметру пунктирной линией, что дает возможность легко позиционировать на них курсор.

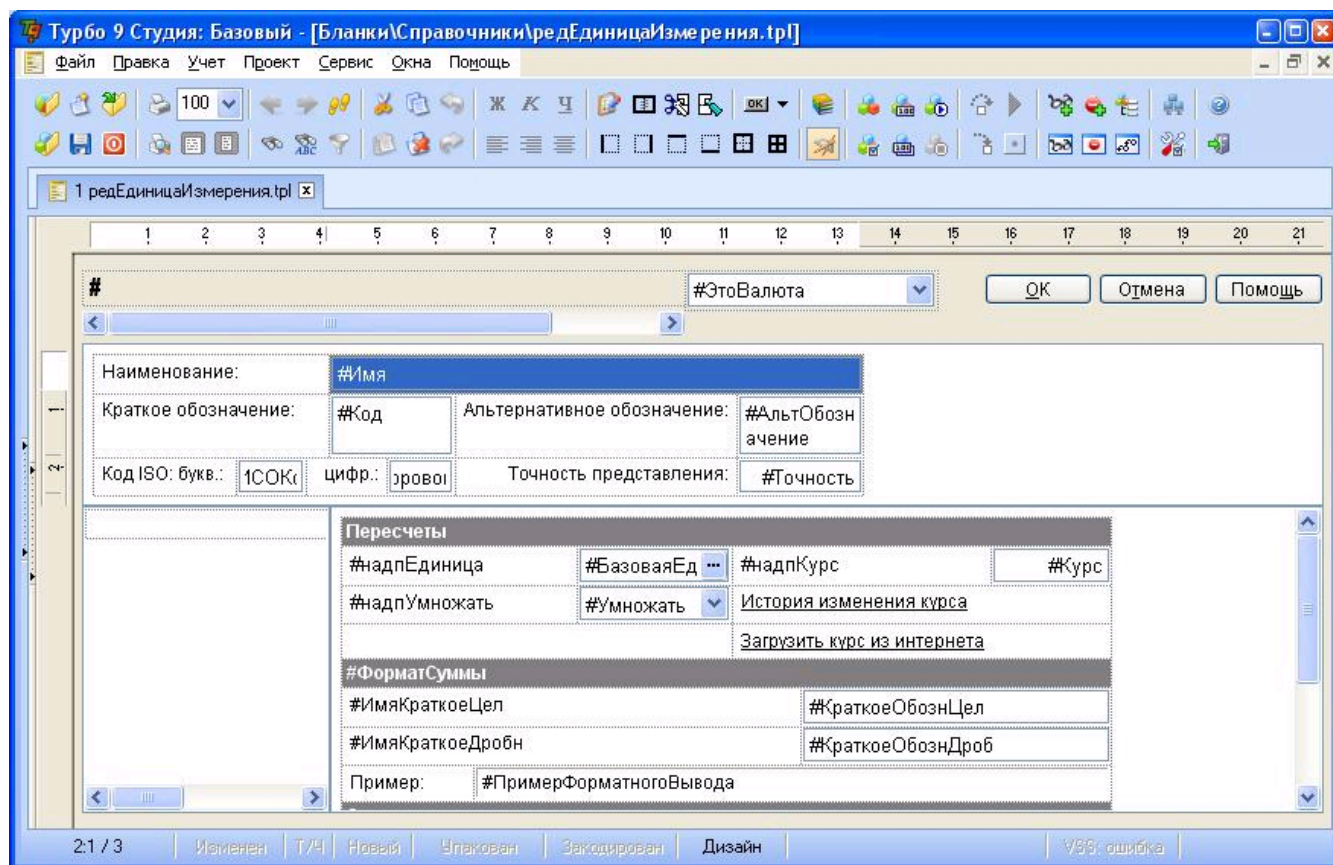


Рис. Окно редактора в дизайн-режиме.

Выделенная клетка (или группа клеток), то есть та, свойства которой в данный момент редактируются, отображается контрастным цветом (по умолчанию, синим), точно также, как это делается в режиме исполнения.

Если клетка связана с переменной или полем записи (в случае бланка-редактора), их имена указываются внутри клетки после знака "решетка" (#). Для изменения этой "привязки" необходимо подвести курсор к требуемой клетке (выделить ее) и нажать *Enter* или же просто выполнить двойной щелчок мышью по клетке. Напомним, что в режиме исполнения в таких клетках выводится значение переменной (поля). Разумеется, в клетке допускается написать и простой текст (без знака # в начале) – тогда он будет выводиться на бланке как есть.

Важную роль в окне редактора играют вертикальная и горизонтальная линейки, которые находятся соответственно слева и сверху над рабочим полем шаблона. На линейках нанесены метки (через каждый сантиметр), позволяющие визуально определять размер тех или иных элементов шаблона. Кроме того, линейки предназначены для выделения столбцов и строк, а также изменения размеров клеток. Для того чтобы выделить столбец, необходимо щелкнуть мышью по горизонтальной линейке непосредственно над столбцом. Курсор при этом превращается в черную вертикальную стрелку. Если требуется выделить несколько соседних столбцов, достаточно нажать левую кнопку мыши, предварительно подведя ее к линейке над столбцом, и затем, не отпуская кнопку, вести курсор вдоль линейки – при этом будут выделяться соответствующие столбцы. Аналогично выделяются строки.

Когда на шаблоне выделена какая-либо клетка, столбец или строка, то на обеих линейках появляется поле, обозначающее размеры текущей секции (той самой, в состав которой входит вышеупомянутая клетка, столбец или строка). Для определенности такие поля (на линейках) будем называть *линейными*. Наведя курсор мыши на границу линейного поля, пользователь может изменить размер секции методом Drag'n'Drop ("перетаски и оставь"). Курсор при этом меняет форму на две параллельных линии, от каждой из которых в противоположные стороны расходятся тонкие стрелки. Внутри линейных полей границы клеток одной и той же секции разделены тонкими серыми рисками. "Потаскив" такую риску с помощью мыши, пользователь изменит размер конкретного столбца или строки.

Что же касается объектов, то они отображаются в дизайн-режиме почти точно также, как и в режиме исполнения. Некоторые отличия зависят от типа объекта (например, в списках не выводятся элементы). Выделенный объект помечается с помощью восьми маленьких квадратов, размещенных в углах и в середине каждой стороны объекта. С помощью данных меток пользователь имеет возможность легко изменить размер объекта – для этого достаточно навести курсор мыши на метку (курсор при этом изменит форму на направленные в две противоположных стороны стрелки), нажать кнопку мыши и,

не отпуская ее, перемещать мышь до тех пор, пока размер не изменится до желаемого значения.

Шаблон бланка или картотеки, т.е. форма, реализующая его внешний вид, в каждый момент времени может находиться в одном из трех состояний:

- в режиме разработки, называемом также *дизайн-режимом*;
- в режиме заполнения полей бланка;
- в режиме предварительного просмотра, аналогичном предыдущему режиму с той лишь разницей, что введенная информация не попадает в переменные бланков.

Внешние отличия перечисленных режимов заключаются в изменении цвета фона шаблона: в дизайн-режиме фон имеет белый цвет, в режимах предварительного просмотра и заполнения - желтоватый. Однако следует иметь в виду, что указанные цвета являются лишь используемыми по умолчанию. Например, разработчик прикладного проекта может задать другие цвета для отображения бланков в режимах просмотра и заполнения.

В дизайн-режиме допустимо изменение параметров шаблона, добавление и удаление входящих в него элементов, изменение правил поведения бланка. В режиме заполнения полей сам шаблон не меняется, но пользователь может редактировать состояние полей, создавая тем самым экземпляры конкретных документов. Режим предварительного просмотра применяется для контроля внешнего вида бланка в процессе его создания.

На стадии проектирования бланка переключение между режимами разработки и предварительного просмотра может быть произведено в любой момент с помощью команды **Дизайн-режим**. Режим заполнения полей автоматически включается во время выполнения проекта на стадии его отладки или в сессии.

В стандартной поставке Студии данная команда входит в контекстное меню бланка, а также присутствует на закладке "Шаблон" инструментальной панели.

Визуальный редактор бланков может быть использован как для создания новых, так и для внесения изменений в уже существующие бланки.

Новый шаблон создается при регистрации бланка в проекте с помощью команды [Добавить](#), контекстного меню, доступной, когда курсор в окне [редактора проекта](#) установлен на ветви Бланки. В результате вызывается Мастер создания бланка, и пользователь может, по своему усмотрению, начать разработку с абсолютно пустого шаблона или любой из нескольких стандартных заготовок.

В процессе работы прикладного проекта бланки открываются в режиме заполнения полей с помощью соответствующих операторов на языке ТБ.Скрипт или же по командам пользователя. Например, в стандартном интерфейсе Студии имеется пункт меню **Учет|Открыть Бланк**. Однако в конкретном прикладном проекте данное меню может быть отключено или модифицировано.

Каждая клетка секции имеет следующий набор свойств:

- *шрифт*, которым выводится текст, отображаемый в клетке. Данное свойство включает в себя гарнитуру шрифта, размер, начертание (полужирные и/или курсив), цвета для фона и текста;
- *тип выравнивания* выведенного текста - по левой границе клетки, по правой границе или по центру;
- *способ вывода*, определяющий поведение клетки в случаях, если выводимый в ней текст превышает размеры строки. Такой текст может быть либо "обрезан" по границе клетки, либо клетка увеличит свою высоту, и текст будет перенесен на следующую строку. Внимание! По умолчанию программа предлагает переносить строки, однако при большом числе клеток с таким свойством отрисовка шаблона может заметно замедлиться;
- *оформление в виде бордюра*, который выводится вокруг поля при необходимости;
- *тип поля*, определяющий, является ли данная клетка полем и для вывода переменной какого типа она используется;
- *режим работы поля* - только вывод значений или редактирование;
- *формат вывода*, определяющий особенности отображения в полях значений числового типа или типа "Дата";
- *текст подсказки*, возникающий на экране, если указатель мыши задержится над данной клеткой;
- *набор процедур и функций* для обработки событий;
- *дополнительные установки*, задающие, например, вывод кнопки обзора (с многоточием) у правого края поля и другие параметры специфических типов полей.

Большинство из вышеперечисленных характеристик клетки (шрифт, выравнивание, способ вывода) могут быть объединены в понятие стиль клетки. Настройка свойств клетки производится в специальном [диалоге](#).

Для шаблона бланка или картотеки, а также для расположенных на нем элементов, определен ряд ситуаций, в которых разработчик электронного документа может воспользоваться так называемыми *событиями*, то есть уведомлениями об изменении состояния программы или действиях пользователя. Это часто требуется для того, чтобы произвести некоторые ответные операции по обработке информации. К подобным ситуациям, например, относятся открытие и закрытие бланка или начало и завершение редактирования поля. Важно понимать, что события генерируются самой системой и, в некотором смысле, представляют собой обратные связи между ней и программой.

Все ситуации, для которых в Студии предусмотрено такое интерактивное взаимодействие, называются *событиями*. Каждому событию может быть назначен *обработчик* — процедура или функция текущего бланка, которая будет вызываться при возникновении соответствующего события. Чтобы быть обработчиком того или иного события, процедура или функция должна иметь заранее определенный прототип, то есть количество и типы аргументов, а также тип возвращаемого значения (в случае функции). Прототип для каждого события свой. Все прототипы обработчиков событий приведены в разделах, относящихся к конкретным классам из [иерархии встроенных классов ТБ.Скрипт](#).

Свои наборы событий имеются как у шаблона в целом, так и у каждой его клетки, которая служит полем ввода или вывода. Для каждого поля можно задать свои обработчики. Также события возникают и для интерфейсных элементов, расположенных на шаблоне (кнопка, флаг, картинка и т.д.), которые называются объектами и рассматриваются в следующих разделах.

Объекты, представляющие собой элементы управления и предназначенные для интерактивного взаимодействия пользователя с программой, не следует путать с объектами программных классов, реализуемых внутри программы на языке ТБ.Скрипт.

Прежде чем переходить непосредственно к описанию механизма работы обработчиков следует напомнить, что для организации взаимосвязи объектов, а также клеток, строк, столбцов и секций шаблона с исходным кодом используется принцип объявления в коде бланка одноименных переменных соответствующих классов (например, переменная класса **СекцияШаблона** будет связана с одноименной секцией на самом шаблоне). Такие переменные инициализируются системой ссылками на элементы шаблона и позволяют обращаться к свойствам и методам встроенных классов ТБ.Скрипт, программно управляющих поведением интерфейсных элементов шаблона. Обратная связь элементов шаблона с исходным кодом как раз и осуществляется с помощью событий.

Для назначения обработчиков событий того или иного объекта, а также шаблона в целом, столбца, строки, клетки или секции, используется страница "События" диалога свойств соответствующего элемента. Вызов диалога свойств осуществляется по команде **Свойства...**, которая обычно доступна из контекстного меню. (Конкретные приемы работы с диалогами свойств, которые специфичны для каждого класса элементов, приводятся в соответствующих разделах.)

На странице "События" данного диалога находится перечень всех событий выбранного элемента шаблона, причем в левой колонке перечня выводится название события, а в правой разработчик может указать имя процедуры или функции - обработчика события. Назначать обработчики можно не для всех событий, а лишь для их части, по необходимости. Название обработчика событий может быть произвольным (но должно соответствовать правилам составления идентификаторов ТБ.Скрипт). Один и тот же обработчик может быть назначен сразу нескольким однотипным элементам (например, одна процедура для отслеживания нажатия любой кнопки бланка).

Для удобства разработчика Студия позволяет автоматически сгенерировать прототип обработчика события и вставить его в исходный код. Для этого достаточно сделать двойной щелчок мышью в клетке, куда предполагается добавить событие. В результате, в `cod`-файле создается заготовка процедуры или функции (в зависимости от требуемого синтаксиса) с именем, включающим название элемента, где происходит событие, и название самого события. Например, если создается обработчик события **ПослеВставки (AfterInsert)** для секции **ТоварныеПозиции**, система сгенерирует для него имя **ТоварныеПозицииПослеВставки**. Причем система пытается проанализировать, на каком языке написано название объекта и добавить название события на нем же. Например, если секция называлась бы **Goods**, то обработчик события получил бы имя **GoodsAfterInsert**. Если у секции не задано имя, вместо него используется строка **Section**. По аналогии обработчики событий в клетках (если клетки не связаны с какой либо переменной или полем) всегда начинаются со строки **Cell**. Для тех клеток, с которыми связана переменная или поле, программа генерирует имя обработчика, начинающееся со строки **Поле (Field)** и имени переменной (поля).

По умолчанию в названии обработчика имена объекта и события пишутся слитно, однако имеется возможность автоматически разделять их символом подчеркивания, для чего в настройках редактора шаблонов на странице "События" имеется флаг **Разделять имя объекта и события символом "_"**.

Автоматически генерируемые заготовки обработчиков событий имеют понятные имена параметров (если они есть) и, в некоторых случаях, пояснительные комментарии. Если в классе-предке уже есть обработчик данного события, то в код заготовки добавляется оператор **inherited/унаследованный**.

По умолчанию заготовка обработчика вставляется в конец `cod`-файла (перед последней компилируемой строкой, как правило это оператор **end**, парный заголовку класса), но разработчик может явным образом задать позицию, куда следует добавлять события с помощью специального комментария "--}". Если Студия находит такую последовательность символов в исходном тексте, то новый обработчик будет вставляться перед данной строкой.

Необходимо иметь в виду, что при работе с обработчиками событий на логику поведения Студии оказывают установки, сделанные в [настройках](#) на странице "События". В частности, автоматическая генерация прототипов событий и их вставка в исходный код происходит только в том случае, если установлен флаг **Разрешить редактирование текста инспектором объектов**. Другой флаг предписывает, нужно ли в генерируемые прототипы обработчиков событий вставлять комментарии. Кроме того, на генерацию прототипов обработчиков влияет и установка языка компилятора, сделанная в настройках ТБ.Скрипт на [странице "Компилятор"](#).

В зависимости от того, какой язык выбран из списка **Основной язык**, программа будет генерировать процедуры и функции с ключевыми словами (операторами, названиями типов) либо на русском, либо на английском языках. Например, когда включен английский язык, то для события **ПриВходе** кнопки Кнопка1 будет сформирован обработчик с операторными скобками **proc...end**, а когда русский - **проц...конец**.

Если программист меняет в окне свойств объекта (секции, столбца, клетки, кнопки и пр.) его название, то программа автоматически корректирует названия имеющихся обработчиков событий этого объекта.

Также если один и тот же обработчик назначен сразу для нескольких объектов, то смена его имени в свойствах одного объекта автоматически приводит к синхронному переименованию обработчиков для других объектов.

Следует отметить, что в диалоге свойств на странице "События" любая клетка, отведенная для ввода имени обработчика, имеет выпадающий список, в котором перечисляются идентификаторы всех методов, чей прототип совпадает с требуемым синтаксисом соответствующего обработчика. Такие списки заполняются на основе информации компиляции, поэтому они остаются пустыми до тех пор, пока исходный текст `cod`-модуля не откомпилирован. После компиляции в качестве обработчика можно выбрать из списка один из имеющихся методов с подходящим прототипом. В частности, в списке совместимых обработчиков событий перечисляются также и подходящие обработчики из классов-предков.

После того как обработчик события назначен, разработчик имеет возможность быстро находить его в исходном коде непосредственно из диалога свойств. Для этого опять же достаточно сделать двойной щелчок мышью на имени требуемого обработчика. В результате открывается `cod`-файл класса и текстовый курсор в нем устанавливается на начало обработчика. Если обработчик определен не в текущем классе, а в одном из предков, то открывается файл соответствующего класса-предка.

Для отмены обработки события необходимо очистить клетку с именем его обработчика.

При реализации распределенных многопользовательских проектов следует иметь в виду, что любые события (будь то события, связанные с шаблоном, картотекой или другими объектами, генерирующими события) происходят только на том клиентском компьютере, на котором выполнены вызвавшие события действия. Другими словами, событие - это программное уведомление об интерактивных действиях локального пользователя или их последствиях.

В клетке секции могут отображаться следующие виды информации:

- статический текст, т.е. текст, помещенный в клетку в процессе разработки шаблонов. Статический текст не может меняться в процессе работы бланка;
- *поле ввода*, с которым связана переменная бланка. В такую клетку информация может быть введена пользователем с клавиатуры;
- *поле вывода*, в котором отображается содержимое переменной бланка без возможности его изменения.

Поля ввода и вывода вместе называются далее *полями* бланка.

Связь между клеткой секции и переменной бланка производится по имени последней: в дизайн-режиме в данную клетку записывается имя, соответствующее переменной бланка и предваряемое символом "#". С полями шаблона могут быть связаны только переменные из текущего бланка или поля записи, для которой текущий бланк является [бланком-редактором](#).

Пусть, например, в коде описана переменная:

```
Дата1: Дата;
```

Тогда поле шаблона можно связать с данной переменной, указав в соответствующей клетке строку:

```
#Дата1
```

Еще раз отметим, что если бланк, использующий шаблон, является бланком-редактором картотеки, то в его полях может также отображаться информация из текущей записи картотеки, в том числе и полученная с помощью ссылок на другие [записи](#). Например:

```
#Корреспондент.Название
```

Здесь **Корреспондент** - название ссылочного поля, содержащего указатель на документ предопределенного типа, скажем, "ЮрЛицо", который в свою очередь имеет поле **Название**.

Кроме того, поля бланка предоставляют дополнительные возможности для вывода на экран информации из [переменных-массивов](#).

Стиль клетки - это совокупность настроек ее свойств (типа выравнивания текста, шрифта, его размера и начертания, цвета символов и фона, тип поля и др.). Стиль клетки может быть сохранен под тем или иным именем в *библиотеке стилей* шаблона.

Каждый бланк может быть оформлен с помощью библиотеки стилей, содержащей стиль для текста самого бланка, а также произвольное число стилей различного назначения: для входных и выходных полей, итоговых сумм и т.д.

При настройке свойств конкретной клетки или группы клеток можно выбрать стиль из библиотеки, и все параметры данной ячейки будут установлены в соответствии с ним.

Свойства, установленные для клетки с помощью выбора стиля, могут быть *переопределены*, т.е. будут чем-то отличаться от того набора характеристик, который диктуется стилем. Переопределенные параметры имеют приоритет перед задаваемыми стилем значениями, и в случае изменения стиля они меняться не будут.

Например, для поля "Итоговая сумма" использован стиль "Поле вывода" и дополнительно к нему изменен цвет фона на красный. При изменении параметров данного стиля в библиотеке в указанной клетке они изменятся все, за исключением цвета фона.

Использование библиотеки стилей позволяет ускорить процесс создания шаблона бланка, а также помогает выдержать единообразие в оформлении документа. Для изменения свойств однородной группы клеток, имеющих один стиль (например, всех полей ввода одного бланка), достаточно изменить его в библиотеке стилей. Подробнее управление библиотекой стилей рассмотрено в разделе [Настройка библиотеки стилей](#).

Для управления внешним видом и поведением шаблона на программном уровне в языке ТБ.Скрипт используется специальный механизм связывания элементов шаблона с программными объектами, основанный на правиле соответствия имен и типов. В дополнение к простым типам в ТБ.Скрипт введены объектные типы (или классы), способные взаимодействовать с интерфейсными элементами шаблона (секциями, строками, столбцами и объектами, встроенными в шаблон - кнопками, флагами, переключателями и т.п.).

В иерархии классов ТБ.Скрипт присутствуют классы **Кнопка**, **Флаг**, **Переключатель**, **Надпись**, **Редактор**, **Список**, **Рисунок**, **Рамка**, **Проигрыватель**, **OLEКонтейнер**, **График**, **КартотекаШаблона**, **ДеревоКартотеки**, **ПодтаблицаШаблона**, производные от класса **ОбъектШаблона**, а также классы **КлеткаШаблона**, **СекцияШаблона**, **СтолбецШаблона**, **СтрокаШаблона**, **Шаблон** (см. [Справочнике по объектному языку ТБ.Скрипт](#)).

По сути дела вышеперечисленные классы представляют собой встроенные типы данных Студии, описывающие ссылку на объект, включенный в состав шаблона. Для установления соответствия между переменной объектного типа и самим объектом необходимо, чтобы имя переменной совпадало с именем объекта, заданным на странице "Общие" окна настройки свойств, а её тип - с типом (классом) этого объекта. Объектные переменные, связанные с шаблоном, должны быть описаны в разделе [InObject](#).

Если для объекта нет такой переменной, то управление им из процедур и функций бланков невозможно; если же существует переменная, не соответствующая ни одному из имеющихся объектов, то эта переменная считается неинициализированной, и все обращения к ней будут приводить к ошибкам.

Внимание! Использование переменных классов, производных от класса **ОбъектШаблона**, а также связанных с секциями, допустимо только тогда, когда бланк открыт на экране.

Пример описания переменных объектных типов:

```
ФлагПроводить :Флаг;  
КнопкаВвода :Кнопка;  
Видеоклип :Проигрыватель;
```

В данном примере предполагается, что в шаблон бланка включены по крайней мере три объекта: флаг, в свойствах которого указано имя **ФлагПроводить**, кнопка с именем **КнопкаВвода** и проигрыватель, которому присвоено имя **Видеоклип**. В этом случае приведенное выше описание позволяет обращаться к этим объектам из процедур и функций данного бланка как к обычным переменным.

Все элементы шаблона (секции, столбцы, строки, элементы управления) должны иметь уникальные (не дублирующиеся) имена. Следует избегать ошибок, связанных с тем, что тип объектной переменной, описанной в cod-файле, отличается от типа одноименного элемента шаблона (например, в шаблоне есть столбец Column1, а в cod-файле введена переменная Column1 типа **TemplateRow**, хотя она должна была бы иметь тип **TemplateColumn**).

Каждый объект обладает набором свойств - переменных, входящих в его состав, и *методов* - процедур и функций. Свойства отражают значения тех или иных параметров объекта, а методы позволяют их изменять и выполнять характерные для объекта операции. Совокупность свойств и методов и есть объект.

Свойства и методы имеют собственные имена и записываются через точку после идентификатора объектной переменной:

```
ФлагПроводить.Разрешен  
КнопкаВвода.Надпись  
Видеоклип.Play
```

На этапе компиляции Студия проверяет правильность записи объектных переменных и их атрибутов, то есть факт наличия конкретного свойства или метода у данного объекта. Кроме того, при использовании свойств и методов в составе выражений или в операторе присваивания система проверяет соответствие используемых типов переменных и параметров, руководствуясь описанием классов (см. [Справочник по объектному языку ТБ.Скрипт](#)). В случае выявления каких-либо расхождений на этапе компиляции генерируется ошибка.

Важно отметить, что некоторые свойства в свою очередь могут иметь объектный тип. Например, в объекте класса **СекцияШаблона** существует свойство **Клетка**, которое, фактически, является двумерным массивом объектов класса **КлеткаШаблона**. Клетка имеет свойство **Шрифт** - объект класса **Шрифт**, обладающий набором собственных свойств. В результате, для того чтобы поменять начертание шрифта конкретной клетки секции шаблона можно, например, записать:

```
Шаблон.ТекущаяСекция.Клетка[3,1].Шрифт.Жирный = ИСТИНА;
```

Здесь назначается жирный шрифт третьей по горизонтали и первой по вертикали клетке текущей секции шаблона. Первое слово "Шаблон" в приведенной строке - тоже имя свойства, на сей раз - в классе **ФормаБланка**; свойство это, как ясно из его названия, содержит ссылку на используемый бланком шаблон.

При обработке исходного текста программы на ТБ.Скрипт проверяется не только наличие у объекта указанных свойств и методов, но и формальная правильность записи обращений к ним. Анализ сочетания имени переменной объектного типа и следующего за ним через точку идентификатора происходит по следующим правилам:

- если указанное сочетание используется в левой части оператора присваивания, то соответствующее имя считается именем свойства, а сама операция - изменением значения свойства;
- если аналогичная конструкция записывается в составе выражения, то она также считается идентификатором свойства объекта и используется для извлечения текущего значения свойства;
- если данное сочетание используется вне выражения и не является левой частью оператора присваивания, то оно рассматривается как вызов метода соответствующего объекта, причем данный метод не должен иметь параметров вызова, т.е. быть процедурой;
- в условиях, аналогичных рассмотренным в предыдущем варианте, после имени метода может следовать открывающая скобка. Такая запись интерпретируется как вызов метода объекта с параметрами, то есть функции.

Примеры вышеперечисленных описаний:

```
-- присвоение свойству объекта значения:
Флаг.Разрешен = ИСТИНА;
-- получения значения свойства у объекта:
НадписьНаКнопке = КнопкаВыхода.Надпись;
-- вызов метода объекта без параметров:
Список.Очистить;
Видеоклип.Play; -- запуск проигрывателя мультимедиа-файлов
Видеоклип.Stop; -- остановка проигрывателя
-- вызов метода объекта с одним параметром строкового типа, содержащим имя файла:
Рисунок.Загрузить ( "LOGO.BMP" );
```

Использование перечисленных выше возможностей позволяет управлять внешним видом и поведением объектов, расположенных на шаблоне, и проектировать гибкие по своему поведению экранные формы, управляемые как внутренними данными, так и действиями пользователя.

Приведем пример простейшего бланка, способного проигрывать мультимедийные файлы. Для этого создадим и откроем новый бланк, перейдем в дизайн-режим и разместим на его шаблоне два объекта: проигрыватель (изображается в виде белого прямоугольника) и кнопку:

После этого настроим основные свойства этих объектов. Для проигрывателя это будут имя объекта (в нашем случае - **Player1**) и имя файла для проигрывания:

Для кнопки следует также задать имя (назовем ее просто **Кнопка**) и надпись на ней.

Кроме того, для кнопки следует задать имя процедуры, которая будет исполняться при нажатии на нее (в терминах Студии - при возникновении события **ПриНажатии**). Для этого нужно перейти на страницу "События" окна настройки свойств кнопки и в строку, соответствующую событию **ПриНажатии**, записать имя будущей процедуры-обработчика. В нашем случае такая процедура будет называться **Управление**:

Процедура **Управление**, вызываемая по нажатию кнопки, должна будет производить следующие действия: если на кнопке написано "Пуск", то запустить проигрыватель и поменять надпись на кнопке на "Стоп"; в противном случае (т.е. когда на кнопке уже написано "Стоп") - остановить исполнение мультимедиа-файла и восстановить надпись "Пуск" на кнопке.

Вот как выглядит этот алгоритм на языке ТБ.Скрипт:

```
Бланк "Проигрыватель";
-- объявляем объекты, чтобы использовать их далее
-- в процедуре Управление
Player1 :MediaPlayer;
Кнопка: Button;
Proc Управление(Об:String);
-- сравниваем свойство объекта "Кнопка" и строку "Пуск",
-- используя функцию приведения типов:
if Кнопка.Надпись = "Пуск" :
-- на кнопке действительно написано "Пуск"
-- запускаем проигрыватель, вызывая его метод Play:
Player1.Play;
-- меняем надпись на кнопке на "Стоп".
-- Функции приведения типов в данном случае не требуется:
Кнопка.Надпись = "Стоп";
else
-- условие не исполнено, следовательно, на кнопке написано "Стоп"
```

```
-- останавливаем проигрыватель, пользуясь его методом Stop
Player1.Stop;
-- восстанавливаем на кнопке надпись "Пуск":
Кнопка.Надпись = "Пуск";
end;
Конец
```

В программе используется механизм регистрации хозяйственных операций в [журналах](#) с помощью специальных записей, также называемых проводка и полупроводка.

Механизм использования полупроводок обусловлен тем, что на внутреннем уровне все движения средств (вне зависимости от типа журнала) представлены в виде полупроводок. Под *полупроводкой* понимается изменение содержимого одного счета (в отличие от проводки, которая затрагивает сразу два корреспондируемых счета), т.е. движение по дебету или кредиту счета с указанием значений параметров, описанных в типе этого счета. *Проводка* - это неразрывная пара двух взаимосвязанных полупроводок (дебетовой и кредитовой), реализующая двойную запись.

Однако, поскольку работа с полупроводками скрыта от конечного пользователя, с его точки зрения программа оперирует проводками. Вместе с тем, в некоторых случаях, когда возникает необходимость изменить счет без корреспонденции, разработчик прикладного проекта имеет возможность разрешить *использование полупроводок в явном виде*.

Концепция использования и синтаксис записи проводок и полупроводок рассматриваются в следующих темах:

- [Синтаксис описания полупроводок](#)
- [Синтаксис описания проводок](#)

Полупроводка - это движение только по одному счету (по дебету или кредиту) с указанием значений параметров, описанных в [типе этого счета](#). В полупроводке указывается идентификатор счета, признак того, является ли счет дебетуемым или кредитуемым, а также перечень дополнительных параметров, в том числе, как правило, суммы. Т.е. полупроводка характеризуется следующими атрибутами:

- Счет;
- Признак того, является ли счет дебетуемым или кредитуемым;
- Список фактических значений параметров.

Существует два варианта употребления полупроводок. Это [текстовые журналы](#) и [типовые операции](#). [Табличные](#) и [картотечные](#) журналы не требуют языкового представления проводок и полупроводок, так как в их случае используется либо встроенный специальный редактор, либо созданные прикладным программистом формы картотек и бланков.

В текстовых журналах и типовых операциях применим следующий синтаксис:

\$Полупроводка	=	(Увеличить Уменьшить) ИмяПланаСчетов "." ИмяСчета [СписокПараметров] КонецСтроки.
\$Увеличить (в типовых операциях)	=	"Дебет" "Debet" .>
\$Увеличить (в текстовых журналах)	=	":" { " " } "+" .
\$Уменьшить (в типовых операциях)	=	"Кредит" "Credit" .
\$Уменьшить (в текстовых журналах)	=	":" { " " } "-" .
\$ИмяСчета (в типовых операциях)	=	Идентификатор ","
\$ИмяСчета (в текстовых журналах)	=	Идентификатор
\$СписокПараметров	=	[ФактическийПараметр] { ["," ФактическийПараметр] } .
\$ФактическийПараметр	=	ИмяПараметра "=" ЗначениеПараметра .
\$КонецСтроки (в типовых операциях)	=	"," [-- комментарий]
\$КонецСтроки (в текстовых журналах)	=	[-- комментарий]

Первый обязательный параметр процедур [Дебет](#) и [Кредит](#) имеет тип **Счет**, фактическое значение записывается непосредственно после имени процедуры и не предваряется именем формального параметра, как все остальные (т.е. получается как бы комбинированный - "позиционно/поименованный" способ передачи параметров).

Остальные параметры передаются стандартным поименованным способом. Имена формальных параметров должны совпадать с именами параметров счета, описанными в [структуре учета](#), также должен совпадать тип фактического параметра и тип параметра счета.

Параметры, которые имеют значение по умолчанию, могут быть опущены. Остальные параметры должны быть указаны обязательно.

В текстовом журнале вся полупроводка должна быть записана целиком на одной строке.

Пример:

```
:+ Склад.Остаток Сумма=60^руб, Количество=10^шт, Товар=Пиво.Балтика.6  
:- Склад.Остаток Сумма=100^usd, Количество=10^ящ, Товар=Пиво.Балтика.9
```

Проводка - это неразрывная пара двух взаимосвязанных [полупроводок](#) (дебетовой и кредитовой), реализующая двойную запись. Она имеет набор атрибутов, которые описываются в [структуре учета](#):

- Счет дебета;
- Счет кредита;
- Список фактических значений параметров, соответствующих формальным параметрам счетов.

В том случае, если оба счета имеют одинаковые параметры и если параметры имеют одинаковый тип и равное значение, то возможно однократное указание имени параметра и его значения. В противном случае необходимо перед именем параметра с помощью знаков "+" или "-" указать, к какому счету он относится. Плюс означает параметр, относящийся к дебетуемому счету, а минус - к кредитуемому.

Существует два варианта употребления проводок. Это [текстовые журналы](#) и [типовые операции](#). [Табличные](#) и [картотечные](#) журналы не требуют языкового представления проводок и полупроводок, так как в их случае используется либо встроенный специальный редактор, либо созданные прикладным программистом формы картотек и бланков.

В текстовых журналах и типовых операциях применяется следующий синтаксис:

```
$СинтаксисПроводки    = Проводка [ИмяПланаСчетов "."] ИмяСчетаДеб  
                      [ИмяПланаСчетов "."] ИмяСчетаКре  
                      [СписокПараметров] КонецСтроки.
```

```
$Проводка              = "Проводка" | "Transaction" .  
(в типовых операциях)
```

```
$Проводка              = ":" .  
(в текстовых журналах)
```

```
$ИмяПланаСчетов        = Идентификатор .
```

```
$ИмяСчетаДеб           = ИмяСчета .
```

```
$ИмяСчетаКре           = ИмяСчета .
```

```
$ИмяСчета              = Идентификатор " , "  
(в типовых операциях)
```

```
$ИмяСчета              = Идентификатор  
(в текстовых журналах)
```

```
$СписокПараметров      = [ФактическийПараметр]  
                      [ { " , " ФактическийПараметр } ] .
```

```
$ФактическийПараметр   = [ "+" | "-" ] ИмяПараметра "="  
                      ЗначениеПараметра .
```

```
$КонецСтроки           = " ; " [ -- комментарий ]  
(в типовых операциях)
```

```
$КонецСтроки           = [ -- комментарий ]  
(в текстовых журналах)
```

Внимание. В текстовом журнале вся проводка должна быть записана целиком на одной строке.

Проводка заменяет пару [полупроводок](#), сохраняя понятие корреспонденции между счетами. Счета записываются двумя первыми обязательными параметрами. Недопустима корреспонденция счетов из разных [планов счетов](#). При этом разрешается опускать имя плана счетов при указании второго счета. Если же имя плана указано в обоих счетах, то оно должно совпадать.

Следует обратить внимание, что в исходном коде типовых операций после идентификаторов счетов ставятся запятые, так же как и после остальных параметров, а в текстовых журналах эти запятые опускаются.

Множество остальных параметров представляет собой объединение множеств параметров обоих счетов. Перед именем формального параметра может идти символ "+" или "-" - это является явным указанием того, к какому

счета - дебета или кредита - относится данный параметр. При этом соблюдаются следующие правила:

1. Если оба счета имеют одноименные параметры и их типы совпадают, то указание в проводке значения этого параметра без символов "+/-", присваивает значение обоим параметрам.
2. Если оба счета имеют одноименные параметры и их типы несовместимы, требуется обязательное раздельное указание значений каждого параметра с явным признаком отнесения к дебету или кредиту (если они не допускают значения по умолчанию). Указание параметра без "+/-" генерирует ошибку.

Пример:

```
: Баланс.01 02 Сумма=99.99^usd, ОС=Станок, Комментарий="Some..."  
: Баланс.46 64 Сумма=1000^руб, +Товар=Пиво.Балтика.6, -Пок=Фирма.Виста
```

Несмотря на то, что синтаксис проводок и полупроводок в текстовых журналах и типовых операциях похож, есть несколько существенных различий. В текстовых журналах запрещено использование функций для вычисления имен счетов и значений аналитических параметров, но зато возможно непосредственное указание любых аналитических признаков. В типовых операциях возможно вычисление счетов и значений любых параметров, но при этом возможно непосредственное указание только [обязательной аналитики](#). Поскольку к моменту компиляции типовых операций нам известны только счета и обязательная аналитика, программа способна выявить конфликты во время компиляции типовых операций.

Для создания прикладных проектов используется объектно-ориентированный процедурный язык ТБ.Скрипт. Именно на языке ТБ.Скрипт записываются алгоритмы обработки данных, причем благодаря его универсальности автоматизации доступны практически все направления деятельности предприятия: это может быть торговля, производство, бухгалтерский или управленческий учет.

Написанные на языке ТБ.Скрипт прикладные программы (проекты) управляют поведением большинства подсистем программы. Например, в состав подсистем и [бланков](#), и [картотек](#) обязательно входит модуль с исходным текстом на языке ТБ.Скрипт, который, собственно, и делает бланки и картотеки полезными, поскольку реализует функционал по обработке информации и взаимодействию с пользователем.

Поскольку проект может быть довольно большим, исходный текст на ТБ.Скрипт, как правило, разделяется на отдельные файлы, каждый из которых имеет расширение *.cod и называется модулем (см. рис. Программирование классов на языке ТБ.Скрипт). Cod-файлы (с исходными текстами на ТБ.Скрипт) редактируются с помощью встроенного [редактора текстов](#). При этом редактор предоставляет специальные [сервисные возможности](#), необходимые для разработчиков.

Очевидно, что деление на модули осуществляется по функциональному принципу, то есть в одном файле записывается набор логически связанных алгоритмов. Это связано не только с необходимостью упростить структуру программы и повысить ее надежность, но с самой идеологией работы подсистем Студии. Так, любой бланк или картотека является в Студии отдельным объектом соответствующего программного класса, что диктуется принципами объектно-ориентированного программирования (ООП).

[Основы программирования](#)

[Стандартные типы данных](#)

[Синтаксис выражений](#)

[Введение в ООП](#)

[Понятие класса](#)

[Модули классов](#)

[Регистрация классов в проекте](#)

[Описание класса](#)

[Заголовок класса](#)

[Наследование и полиморфизм](#)

[Свойства класса и объектов](#)

[Описание переменных](#)

[Константы](#)

[Процедуры и функции](#)

[Операторы](#)

[Директивы языка ТБ.Скрипт](#)

[Препроцессор](#)

[Обработка исключительных ситуаций](#)

[Отладчик](#)

[Информация о типах времени исполнения \(RTTI\)](#)

[Специальные возможности текстового редактора](#)

[Преобразования формата](#)

Операторы

Тело процедуры или функции состоит из операторов - элементарных синтаксических конструкций, комбинируя которые, можно записывать алгоритмы - последовательности выполнения тех или иных действий и расчетов.

Среди операторов выделяются следующие языковые конструкции:

- [Оператор присваивания](#)
- [Оператор вызова процедуры или функции](#)
- [Условный оператор \(if/если\)](#)
- [Оператор выхода \(return/возврат\)](#)
- [Оператор цикла. Виды циклов](#)
- [Целочисленный цикл \(for/для\)](#)
- [Цикл с предусловием\(while/пока\)](#)
- [Вложенные циклы](#)
- [Оператор прерывания цикла\(break/прервать\)](#)
- [Оператор изменения области видимости \(with\)](#)
- [Оператор приведения типов \(as/как\)](#)
- [Оператор проверки совместимости типов \(is/есть\)](#)
- [Оператор проверки наличия значения в массиве](#)

Кроме того, в языке ТБ.Скрипт используются:

- стандартные [арифметические, строковые и логические операции](#), с помощью которых составляются сложные синтаксические выражения;
- [побитовые операции](#) **Not**, **And**, **Or** и **Xor**, которые допустимы *только для целочисленных операндов*.

Оператор цикла - это всего лишь один из видов операторов языка ТБ.Скрипт. Его можно рассматривать как элементарный "кирпичик", из которых строится "здание" алгоритма. Однако внутри оператора цикла между его заголовком и словом КОНЕЦ можно записывать другие операторы, которые, собственно, и будут повторяться. Кроме того, внутри одного цикла можно разместить другой цикл, этого же или другого типа. Такие циклы называются вложенными и находят широкое применение в программировании.

Вложенные циклы позволяют программисту строить сложные алгоритмы обработки информации. Как было рассмотрено при описании различных видов циклов, каждый из них обеспечивает перебор некоторых сущностей (с помощью счетчика или без него) лишь в одном измерении. Иногда возникает необходимость обработать данные более сложной, нежели одномерная, структуры, например, двумерный массив.

В подобных случаях следует использовать вложенные циклы. Вложенные циклы подчиняются общим правилам, по которым реализуются циклы ТБ.Скрипта.

Пример:

```
ПЕРЕМ ЗапросДокументов : Запрос;
ПЕРЕМ ТекущийДокумент : Пример.Накладные;
ПЕРЕМ i : Целое;

-- Строим запрос по накладным
ЗапросДокументов = Запрос.Создать([Пример.Накладные]);
ЗапросДокументов.Выбор;

-- Начинаем внешний цикл по найденным документам.
-- Когда будет достигнут и обработан последний документ,
-- условие этого цикла перестанет выполняться
ПОКА ЗапросДокументов.ВКонец = ЛОЖЬ ЦИКЛ
    ТекущийДокумент = ЗапросДокументов.Текущий;
    -- Начинаем внутренний цикл по позициям текущей накладной
    ДЛЯ i=1..ТекущийДокумент.ПозицииТМЦ.Количество ЦИКЛ
        -- ... обработка товара i из текущей накладной
        trace(ТекущийДокумент.ПозицииТМЦ[i].Наименование);
    КОНЕЦ;
    -- Переходим на следующий документ
    ЗапросДокументов.Следующий;
КОНЕЦ;
```

Если используются вложенные циклы со счетчиками, то на каждой из итераций операторам, записанным внутри таких циклов, доступны оба счетчика. Их можно использовать, например, для адресации к элементам двумерного массива. Таким образом, вложенные циклы перебирают все возможные комбинации из своих счетчиков.

Вложенные циклы могут пригодиться для описания на языке ТБ.Скрипт таких отчетов, как, например, шахматная или оборотно-аналитическая ведомость, где необходимо сформировать и столбцы, и строки таблицы.

Разумеется, ТБ.Скрипт позволяет записывать циклы неограниченного уровня вложенности, однако следует иметь в виду, что большой уровень вложенности приводит к замедлению работы программы.

Для значений логического типа в ТБ Скрипте введены логические операции:

- логического умножения И (AND);
- логического сложения ИЛИ (OR);
- исключающего ИЛИ (XOR);
- отрицания НЕ (NOT).

Все они также возвращают результат логического типа. Суть каждой операции поясняют следующие таблицы:

Операция ИЛИ

Операнд 1	Операнд 2	Результат
Истина	Истина	Истина
Истина	Ложь	Истина
Ложь	Ложь	Ложь

Операция И

Операнд 1	Операнд 2	Результат
Истина	Истина	Истина
Истина	Ложь	Ложь
Ложь	Ложь	Ложь

Операция XOR

Операнд 1	Операнд 2	Результат
Истина	Истина	Ложь
Истина	Ложь	Истина
Ложь	Ложь	Ложь

Операция НЕ

Операнд	Результат
Истина	Ложь
Ложь	Истина

Таким образом, операция ИЛИ дает значение ИСТИНА, если по крайней мере один из операндов истинен; операция И - если истинное значение имеет и тот, и другой операнд; операция XOR дает значение ИСТИНА, когда только один из операндов имеет значение ИСТИНА; операция НЕ применяется к одному операнду и возвращает обратное ему значение (такую операцию называют еще инвертированием).

При создании сложных программ их удобнее разделить на логические части, каждая из которых относительно невелика по размеру и предназначена для достижения некоторых промежуточных целей. В этом случае облегчаются процессы программирования и отладки (поиски ошибок в программе), повышается наглядность исходных текстов.

Как правило, фрагменты алгоритма оформляют в виде отдельных процедур и функций, каждая из которых выполняет свою часть общей работы. Для того, чтобы "запустить" ту или иную процедуру или функцию из другого метода, можно воспользоваться оператором вызова. Говорят, что одна процедура или функция вызывает другую.

Синтаксис оператора вызова процедуры прост: вызов метода производится путем указания его идентификатора с перечислением в скобках через запятую параметров, если они есть. Параметры могут быть заданы выражениями. В этом случае выражение сначала вычисляется, а потом уже в виде значения передается в процедуру или функцию. Параметры вызова метода подставляются на место формальных параметров, описанных в [заголовке процедуры или функции](#) и участвующих во внутреннем алгоритме.

Примеры вызовов:

```
-- вызов процедуры без параметров:  
Рассчитать;  
-- вызов процедуры с двумя параметрами, первый из которых является выражением:  
НайтиБольшее(Оборот50, 0);
```

Оператор вызова помогает реализовать на практике принцип нисходящего программирования, рассмотренный в [начале данного руководства](#), т.е. дает возможность разделить сложную программу на относительно простые части и комбинировать их, вызывая в нужные моменты.

У некоторых процедур и функций встроенных классов Студии есть необязательные аргументы - такие аргументы можно не указывать при вызове. Для того чтобы различать обязательные и необязательные аргументы, последние указываются в прототипе функции или процедуры в квадратных скобках. Например, процедура **Звук** встроенного класса **Консоль** описана следующим образом:

```
Звук([ТипЗвука:Целое]);
```

Это означает, что ее можно вызвать и с параметром и без, причем в последнем случае в качестве значения ТипЗвука будет браться некоторое стандартное значение, обговариваемое в справке по соответствующей процедуре (функции). Приведенный пример также иллюстрирует и тот факт, что если необязательный аргумент один и его необходимо "опустить", то в операторе вызова также "опускаются" и круглые скобки, задающие список аргументов. Иными словами, если все аргументы являются необязательными, то разрешается не указывать весь список аргументов, включая круглые скобки.

Возможны случаи, когда необязательные аргументы находятся в начале или середине списка аргументов. Тогда при вызове метода место таких аргументов должно быть обозначено запятыми. Например, в классе консоль имеется функция

```
Вопрос ([Заголовок:Строка]; Вопрос:Строка; ТекстКнопок[:Строка]): Целое;
```

Аргумент Заголовок - необязательный. Однако, если его просто не указать, то компилятор ТБ.Скрипт "не поймет" оператора вызова, так как число и типы оставшихся аргументов перестанут соответствовать прототипу. Поэтому для вызова данной функции без первого аргумента следует начать список аргументов с запятой, несмотря на то, что перед ней будет пустое место:

```
Ответ = Вопрос ( , "Как дела?", ВариантыОтветов);
```

Необязательные аргументы, находящиеся в конце списка аргументов можно "опускать" вместе с запятыми. Например, при вызове функции Альтернатива, имеющей прототип

```
Альтернатива (Заголовок:Строка; Строки[:Строка [; Ширина:Целое]): Целое;
```

допускается указать только 2 параметра:

```
Ответ = Альтернатива ("Продолжить процесс", ВариантыОтветов);
```

Аналогично можно "опустить" все необязательные аргументы, последовательно идущие в конце списка аргументов.

Оператор выхода заканчивает работу процедуры или функции и осуществляет переход к той части программы, которая следует за оператором ее вызова. Фрагмент программы, находящийся после оператора выхода, не исполняется.

Оператор выхода использует ключевое слово ВОЗВРАТ (RETURN), после которого в случае функции следует выражение, определяющее возвращаемое значение. Напомним, что функция должна возвращать значение определенного типа.

Оператор ВОЗВРАТ может также обозначаться символом "^", по аналогии с оператором прекращения работы функции в предыдущих версиях Турбо Бухгалтера. Однако этот старый синтаксис применять настоятельно не рекомендуется.

Как правило, оператор выхода имеет смысл использовать внутри тела процедуры или функции, когда требуется прервать последовательное выполнение алгоритма. Писать оператор выхода в конце процедуры бессмысленно, так как завершающее ключевое слово КОНЕЦ (или ЦОРП) само по себе задает возврат управления вызывающему участку исходного кода. В функциях оператор выхода используется для возврата значений, а потому он может находиться в любом месте функции.

Пример использования операторов выхода:

```
ПРОЦ Наоборот(Var Признак:Логическое); -- процедура
    ЕСЛИ Признак = ИСТИНА тогда
        Признак = ЛОЖЬ;
    ВОЗВРАТ;
ИЛСЕ;
    Признак = ИСТИНА;
КОНЕЦ;
ФУНК ВсегдаНоль :ЦЕЛОЕ;
    ВОЗВРАТ 0; -- функция всегда возвращает 0
КОНЕЦ;
ФУНК Сумма2 (Слаг1, Слаг2 :ЧИСЛО) :ЧИСЛО;
    ^(Слаг1+Слаг2); -- функция возвращает сумму двух чисел
КОНЕЦ;
```

В функциях существует неявно описанная локальная переменная **Результат (Result)**. Она имеет тот же тип, что и функция, в которой она описана, и, как и все локальные переменные, в начале работы функции инициализируется пустым значением данного типа. Эта переменная используется для хранения возвращаемого значения функции; именно из нее берется результат работы функции. Т.е. конструкция "return 2" аналогична конструкции "Result = 2". В то же время использование переменной **Result** удобнее, чем оператора Return, поскольку значение **Result** можно изменять несколько раз в процессе работы функции, а также обращаться к переменной **Result** на чтение, чтобы узнать текущее состояние возвращаемого значения.

Вследствие существования переменной **Result** правила использования оператора Return в функциях доопределяются следующим образом:

- можно писать "Return;" вместо "Return <значение>";
- можно вообще не использовать Return в функции.

Если значение переменной **Result** не изменялось, то функция вернет nil (нулевое значение соответствующего типа).

Оператор изменения области видимости **With** предназначен для обращения к свойствам объектов без указания их полного имени, включающего имена объектных типов, разделенных точкой ([операцией разыменования](#)). Оператор состоит из заголовка, тела и завершающего ключевого слова **End**. Заголовок начинается с ключевого слова **With**, после которого идет выражение объектного типа и ключевое слово **Do**. В теле оператора можно записывать любые выражения и другие выражения, использующие свойства того объекта, который упомянут в заголовке, без указания самого объекта. Иными словами, один раз указав имя объекта в заголовке оператора **With**, затем можно работать с его свойствами напрямую.

Синтаксис оператора следующий:

```
With <выражение объектного типа> Do
    <обращения к свойствам объекта>
End;
```

Фактически с помощью оператора **With** создается локальная область видимости для конкретного объекта. Внутри этой области компилятор ТБ.Скрипт позволяет использовать краткие названия свойств, подразумевая их отношение к указанному объекту, то есть как бы в контексте этого объекта. В связи с этим **With** также называют оператором изменения контекста.

Поясним суть данного оператора.

В силу того, что ТБ.Скрипт является объектно-ориентированным языком, в программах, написанных на нем, часто встречаются обращения к свойствам объектов. Например, для изменения параметров запроса, который представлен в языке ТБ.Скрипт объектом одноименного класса, можно написать:

```
Запрос1.Фильтр = "Дата=12.12.1999";
```

Здесь **Запрос1** - это переменная класса **Запрос**, а **Фильтр** - свойство этого класса, имеющее строковый тип.

Иногда само свойство объекта представляет собой объект другого класса, причем уровень вложенности последовательных операций разыменования может быть довольно большим. Например, в следующем фрагменте кода:

```
Документы.Накладная.Позиции[i].Цена
```

происходит обращение к элементам подтаблицы **Позиции** объекта **Накладная**, причем из каждого элемента берется поле **Цена**. В принципе, операция разыменования требует некоторого времени на исполнение, поэтому, если в нескольких соседних строчках программы используются одни и те же обращения к свойствам объектов, имеет смысл выполнить общие операции разыменования только один раз и присвоить результат промежуточной переменной соответствующего типа. Например, цикл:

```
For i=1..Количество do
    Документы.Накладная.Позиции[i].Сумма =
        Документы.Накладная.Позиции[i].Цена *
        Документы.Накладная.Позиции[i].Количество;
End;
```

можно было бы оптимизировать следующим образом:

```
Var Товар:Документы.Накладная.Позиции;
For i=1..Количество do
    Товар = Документы.Накладная.Позиции[i];
    Товар.Сумма = Товар.Цена * Товар.Количество;
End;
```

Для того, чтобы исключить необходимость введения дополнительной переменной, и предназначен оператор **With**. Вышеприведенный пример легко переписать с использованием **With**:

```
For i=1..Количество do
    With Документы.Накладная.Позиции[i] do
        Сумма = Цена * Количество;
    End;
End;
```

Как видно из примера, введение **With** упрощает исходный код. Кроме того, оно увеличивает скорость выполнения программы.

Однако иногда злоупотребление оператором **With** приводит к ухудшению "читабельности" программы. Кроме того, существует несколько нюансов, которые могут послужить источником ошибок при использовании **With**.

Рассмотрим более сложный пример.

```

var Q :Query;
var i :Integer;
Q = Query.Create([Документы.Накладная]);
Q.Select;
while not Q.EOF do
  -- обратите внимания на явное приведение типа
  with Документы.Накладная(Q.Current) do
    for i=1..Позиции.Количество do
      with Позиции[i] do
        Сумма = Цена * Количество;
        Итого = Итого + Сумма;
      end;
    end;
  end;
  Q.Next;
end;

```

Создав запрос по документам типа **Накладная**, мы проходим в цикле **While** по всем отобранным документам и пересчитываем в них в цикле **For** информацию по товарам. Прежде всего следует отметить, что абстрактный тип **Документ**, возвращаемый методом **Current** объекта **Q**, приводится к конкретному типу **Документы.Накладная**. Это означает, что компилятор может проверить правильность обращения к свойствам и методам объекта еще на стадии компиляции. При этом в исходном тексте могут встречаться обращения к глобальным переменным и свойствам классов, не требующих явного указания имени, таких как **Система**, **Бухгалтерия** или **Консоль**. Компилятор в этом случае правильно трактует все идентификаторы.

Следует иметь в виду, что в заголовке **With** нельзя указывать выражение абстрактного типа, например, **Запись**. Сделано это исходя из следующих соображений.

Если бы в заголовке **With** стояло выражение абстрактного типа (например, в нашем случае - **Q.Current**), то внутри оператора **With** можно было бы в качестве вызова свойства или метода записать фактически произвольную строку, поскольку использование абстрактных классов заставляет компилятор ТБ.Скрипта применять так называемую динамическую диспетчеризацию. В этом случае на стадии компиляции еще не известно, значение какого типа будет содержать объект (так, объект класса **Документ** может содержать любой объект производного класса, скажем, **Накладная**, **Счет**, **Приходный Ордер** и т.д.) и поэтому проверка наличия конкретного свойства не проводится. Вместо этого компилятор генерирует код, позволяющий вызвать динамическое свойство по его имени, что фактически эквивалентно отложенной проверке существования свойства до момента исполнения кода. Таким образом, внутри оператора **With** с выражением абстрактного типа в заголовке все идентификаторы интерпретировались бы на стадии исполнения как свойства того самого объекта, что стоит в заголовке оператора. Если в такой ситуации обратиться к методу одного из классов Студии, определенных в глобальной области видимости (и такой метод отсутствует в объекте заголовка), возникла бы ошибка.

Возвращаясь к предыдущему примеру, отметим, что в нем используются два вложенных оператора **With**. В результате этого в теле внутреннего оператора **With** возможно двоякое толкование идентификаторов. Так, в строке

```
Итого = Итого + Сумма;
```

Свойство **Сумма** относится к объекту **Документы.Накладная.Позиции[i]**, а свойство **Итого** - к объекту **Документы.Накладная**. Компилятор "понимает" это легко, однако программист должен быть особенно внимательным.

Если бы объекты внутреннего и внешнего операторов имели свойства с одним и тем же именем, то в теле внутреннего **With** обращение происходило бы к свойству объекта внутреннего **With**.

В случае необходимости любой цикл можно прервать с помощью специального оператора **ПРЕРВАТЬ (BREAK)**. Его допустимо использовать только внутри циклов. Если оператор расположен внутри вложенного цикла, то прерывается лишь один, самый внутренний цикл, содержащий оператор BREAK.

Пример:

```
ПЕРЕМ i,j : Целое;

j = 0;
-- Начинаем внешний цикл
ПОКА ИСТИНА ЦИКЛ
  -- Начинаем внутренний цикл
  ДЛЯ i=1..10 ЦИКЛ
    ЕСЛИ i = 5 ТОГДА
      -- при i = 5 внутренний цикл будет прерван
      ПРЕРВАТЬ;
    КОНЕЦ;
  КОНЕЦ;
  -- цикл по i всегда будет заканчиваться при i=5
  -- ...
  j = j+1;
  -- как только j станет больше i,
  -- прерывается и внешний цикл
  ЕСЛИ j > i ТОГДА
    ПРЕРВАТЬ;
  КОНЕЦ;
КОНЕЦ;
```


Оператор приведения типов

Оператор приведения типов **as (как)** используется, как и следует из его названия, для приведения типов, то есть для преобразования значений одного типа к другому.

Синтаксис оператора следующий:

```
<выражение целевого типа> = <выражение исходного типа> (as | как) <целевой тип>
```

Оператор преобразует выражение исходного типа в выражение целевого типа, если такое преобразование допустимо.

Приведение возможно только между совместимыми типами, например, от перечисления к целому, или от базового типа **Объект** к другому объектному типу, значение которого фактически содержится в значении типа **Объект**. Так, если в переменную типа **ОбъектШаблон** была записана ссылка на объект **Редактор**, то возможно обратное приведение типа для получения редактора из объекта шаблона:

```
proc P1(Элемент: ОбъектШаблона);  
var Ред: Редактор;  
    Ред = Элемент as Редактор;  
    ...  
end;
```

В случае объектных типов их совместимость фактически определяется тем, является ли один из них предком другого, и тесно связана с механизмом наследования.

Если попытка преобразования типов выполняется между несовместимыми типами, на стадии исполнения программы генерируется ошибка. В целях исключения подобных ошибок, как правило, приведение типа предваряют [проверкой совместимости](#).

Преобразование типов обычно используется в целях обобщения алгоритмов, так как позволяет в переменных и параметрах базовых классов (типов) хранить значения различных производных классов и обрабатывать их унифицированными методами.

Оператор присваивания

Оператор присваивания служит для изменения значения переменной бланка или локальной переменной. Форма этого оператора напоминает обычную математическую запись:

Примеры оператора присваивания:

```
Цена = 2.00; -- присваивание значения
Количество = 50;
Сумма = Цена*Количество; -- присваивание выражения
```

Допустимо присваивание значения переменной текущего или другого бланка, а также локальной переменной. В правой части оператора присваивания может стоять выражение, включающее в себя функции, в том числе и стандартные. Например, записав

```
Дата = СЕГОДНЯ;
```

можно присвоить переменной **Дата** значение, возвращаемое стандартной функцией **Сегодня**.

Для присвоения значений [переменным-массивам](#) необходимо явно указывать индекс элемента, значение которого должно быть изменено. В качестве индекса можно также использовать целочисленные переменные, выражения и т.п.

Пример присвоения значений элементам переменных-массивов:

```
Сумма[2] = 12000;
Товар[ВыбранныйНомер] = 'Шелк';
```

В последнем примере в качестве индекса переменной **Товар** использована другая переменная - **ВыбранныйНомер**. Эта переменная должна быть целого типа, т.е. иметь следующее описание:

```
ВыбранныйНомер :ЦЕЛОЕ;
```

Оператор **in** предназначен для проверки наличия в массиве указанного значения.

Синтаксис:

`<выражение> in <массив>`

Если значение выражения содержится в указанном массиве, то оператор возвращает значение Истина. Фактически оператор осуществляет поиск в массиве, аналогично функции `SearchInArray`.

Типы выражения и элементов массива должны совпадать и быть связаны родственными отношениями.

Пример:

```
type ДеньНедели = (днПонедельник, днВторник, днСреда, днЧетверг, днПятница, днСуббота,
днВоскресенье);
ДеньДоступа :ДеньНедели[] = [днПонедельник, днСреда, днПятница];

func РазрешенныйДень(День :ДеньНедели): Logical;
    return День in ДеньДоступа;
end;
```

Данный оператор можно использовать в фильтрах. Здесь его использование более оправданно, чем в языке ТБ.Скрипт, т.к. в ряде случаев он переводится в SQL-выражение, что значительно ускоряет отбор. Подобная оптимизация не может быть выполнена только в случаях, если в составе массива имеются неконстанты или в выражение входит мягкая ссылка (разыменовываемая переменная некоего базового класса, содержащая объект производного класса).

Оператор проверки совместимости типов

Оператор **is** (**есть**) позволяет проверить значение выражения (в простейшем случае, переменной) на принадлежность к некоторому указанному типу.

Синтаксис:

<проверяемое выражение> (**is** | **есть**) <идентификатор целевого типа>

Результатом выполнения оператора является значение логического типа, которое равно TRUE, если проверяемое выражения имеет целевой или производный от целевого тип, и FALSE - в противном случае.

Оператор **is** как правило используется совместно с оператором [приведения типа as](#):

```
proc Pl(V: Variant);
var Money: Unit Валюта;
  if V is Unit Валюта then
    Money = V as Unit Валюта;
  ...
end;
end;
```

Оператор **is** может применяться как для объектных типов, так и для специальных бухгалтерских типов (Счет, Признак).

Если проверяемое выражение представляет собой неинициализированную переменную (параметр), оператор **is** возвращает значение FALSE, не генерируя исключительной ситуации.

Оператор цикла. Виды циклов

Одно из достоинств компьютера заключается в том, что он может исполнять один и тот же набор операторов много раз и достаточно быстро. Поэтому в ТБ.Скрипт была введена конструкция, позволяющая задавать число и условия повторения последовательности операторов. Такая конструкция называется *оператором цикла* или просто *циклом*.

В зависимости от особенностей решаемой задачи в процедурах и функциях можно использовать два вида циклов:

- [Целочисленный цикл](#)
- [Цикл с условием](#)

Синтаксически циклы разных видов отличаются своими заголовками.

В языке ТБ.Скрипт можно использовать побитовые операции **Not**, **And**, **Or** и **Xor**. Эти операции допустимы *только для целочисленных операндов*.

Внимание. Так как в языке ТБ.Скрипт операция сравнения "=" имеет более высокий приоритет по сравнению с побитовыми операциями, то запись вида

```
if vIntVar and 2 <> 0 then {...}
```

приведёт к ошибке компиляции "Несовместимые типы операндов...". Для устранения этой ошибки следует явно выделить побитовые операции скобками:

```
if (vIntVar and 2) <> 0 then {...}
```

Пример использования побитовых операций

```
func ВернутьИмяРеквизита(aСтрока :String) :String;  
  var локТочка      :Integer;  
  var локДвоеточие  :Integer;  
  
  локТочка = pos('.', aСтрока);  
  локДвоеточие = pos(':', aСтрока);  
  if (локТочка = 0) and (локДвоеточие = 0) then  
    Result = aСтрока;  
  elseif (локТочка = 0) xor (локДвоеточие = 0) then  
    Result = SubStr(aСтрока, 1, max([локТочка, локДвоеточие]) - 1);  
  else  
    Result = SubStr(aСтрока, 1, min([локТочка, локДвоеточие]) - 1);  
  end;  
end;
```

Истина | True
Ложь | False
Пусто | Nil
Или | Or
И | And
Хор
Не | Not
Если | If
Конец | End
Импорт | Import
Классы | Classes
Класс | Class
ВКласе | InClass
ВОбъекте | InObject
Публично | Public
Лично | Private
Проц | Proc
Функ | Func
Возврат | Return
Выход | Exit
Перем | Var
Попытка | Try
Окончание | Finally
Исключение | Except
Возбудить | Raise
ЕслиЖе | ElseIf
Иначе | Else
Пока | While
Цикл | Do
Для | For
Прервать | Break
Цорп | Corp
Кнуф | Cnuf
Илсе | Fi
Лкиц | Od
Тогда | To | Then
Бланк | Blank | БланкСШаблоном | BlankWithTemplate
Inherited | Унаследованный
With
Опер | Oper
Self
Как | As
Есть | Is
Синоним | Synonym

Арифметические операции (применимы для типов целое, число):

+	сложение;
*	умножение;
-	вычитание;
/	деление;
%	взятие процента от числа.

Дополнительно для типа строка сложение означает конкатенацию. Определено также сложение и вычитание даты с целым числом и друг с другом.

Операции сравнения (применимы для типов целое, число, строка, дата):

<	меньше;
>	больше;
=	равно;
<=	меньше или равно;
>=	больше или равно;
<>	не равно.

[Логические операции](#) и [побитовые операции](#):

ИЛИ (OR)	логическое сложение;
И (AND)	логическое умножение;
ИЛИ (XOR)	исключающее ИЛИ;
НЕ (NOT)	отрицание.

Внимание. Побитовые операции допустимы *только для целочисленных операндов*.

Для объектных типов определены следующие операции:

=	присваивание ссылки на объект (копирование объекта выполняется процедурой Assign/Присвоить);
.	(точка) разыменование (получение значения свойства по его имени).

Условный оператор служит для управления ходом выполнения алгоритма. В процессе работы с программой довольно часто встречаются такие ситуации, когда какие-либо действия требуется выполнять не всегда, а только при определенных условиях. В таких случаях эти действия рекомендуется записывать в виде набора операторов внутри условного оператора, задающего условия их выполнения.

Один условный оператор может задавать несколько пар "условие - действие". Такая пара называется альтернативой. В условном операторе допустимо до 500 альтернатив, чего, как правило, достаточно для написания алгоритмов любой сложности.

При работе условного оператора сначала проверяется на выполнимость первое условие оператора, которое задается логическим выражением и, соответственно, может принимать либо значение ИСТИНА, либо ЛОЖЬ. Если условие истинно, то исполняются соответствующие ему операторы, а выполнение самого условного оператора заканчивается. В противном случае проверяется на истинность следующая альтернатива, если она есть, и т.д.

Первая альтернатива начинается с ключевого слова ЕСЛИ (IF), а все последующие со слова ЕСЛИЖЕ (ELSIF).

Для записи операций, которые необходимо выполнить в том случае, если ни одна из альтернатив не была исполнена, используется ключевое слово ИНАЧЕ (ELSE).

Конструкцию ИНАЧЕ следует указывать последней в списке альтернатив условного оператора, поскольку она выполняется безусловно (если управление переходит на нее) и поэтому ни одна из альтернатив, стоящих за ней, никогда не будет выполнена.

Заканчивается условный оператор либо ключевым словом КОНЕЦ (END), либо инверсией открывающей оператор директивы - ключевым словом ИЛСЕ (FI).

Пример условных операторов:

```
ЕСЛИ Остаток50 > 0 тогда
    Сообщение('Остаток на 50 счете больше нуля');
КОНЕЦ;
ЕСЛИ Буква = 'А' тогда
    Сообщение('Эта буква - А');
ЕСЛИЖЕ Буква = 'В' тогда
    Сообщение('Эта буква - В');
ИНАЧЕ -- ни одна из вышеперечисленных альтернатив
    Сообщение('Эта буква - ни А, ни В');
;КОНЕЦ

-- пример с использованием англоязычных ключевых слов:
IF Буква = 'А' then
    Сообщение('Эта буква - А');
ELSIF Буква = 'В' then
    Сообщение('Эта буква - В');
ELSE -- ни одна из вышеперечисленных альтернатив
    Сообщение('Эта буква - ни А, ни В');
END;
```

Еще один особый синтаксис оператора if используется при записи его в виде функции, что может быть полезно в случае формирования сложных однострочных выражений.

Синтаксис функции if следующий:

```
[ if | если ] (<ЛогическоеУсловие> , <Выражение1> , <Выражение2>)
```

Когда логическое условие имеет истинное значение, выполняется выражение 1. В противном случае - выражение 2. В обоих случаях результат функции if имеет тип, соответствующий актуальному (выбранному) выражению, то есть не обязательно логический. Функции if могут быть вложены друг в друга, например:

```
D = if (A > B , 1, if(A < B , -1, 0) ); /
```

Целочисленный цикл представляет собой базовую конструкцию для повторения операторов несколько раз. В его заголовке просто задается число необходимых повторов.

Целочисленный цикл начинается с заголовка, идентифицируемого по ключевому слову **ДЛЯ** (англоязычный вариант **FOR**). В конце заголовка, включающего так называемый счетчик - переменную цикла, а также условия его повторения, записывается ключевое слово **ЦИКЛ** (**DO**). Затем следует тело цикла - непосредственно набор операторов, которые требуется выполнить несколько раз. Заканчивается цикл, как и все конструкции языков Студии, или словом **КОНЕЦ** (**END**), или инверсией слова "цикл" - ключевым словом **ЛКИЦ** (**OD**).

Счетчиком цикла может выступать глобальная или локальная переменная. Как правило, для счетчиков используются локальные переменные, поскольку информация, хранящаяся в них, является временной.

Общий синтаксис заголовка оператора следующий:

```
for <счетчик> = <Начальное значение>..<Конечное значение> [step <Размер шага>] do
```

Аналогичная запись с русскими синонимами ключевых слов:

```
для <счетчик> = <Начальное значение>..<Конечное значение> [шаг <Размер шага>] цикл
```

Счетчик целочисленного цикла, как следует из его названия, должен иметь тип **ЦЕЛОЕ**. В нем хранится определенное число, которое изменяется (увеличивается или уменьшается) при каждом повторении цикла, называемом итерацией. В заголовке целочисленного цикла указываются начальное и конечное значения счетчика, а также, при необходимости, и шаг, на который счетчик будет изменяться на каждой итерации. По умолчанию, если шаг не указан, он считается равным единице. В качестве шага допускается указывать любое целое число - как положительное, так и отрицательное. Процесс повторения цикла происходит до тех пор, пока значение счетчика находится между начальным и конечным значением (включительно). Если шаг положителен (или опущен, то есть равен 1), то цикл завершается при превышении счетчиком конечного значения. Если же шаг отрицательный, то конечное значение должно быть меньше начального и, соответственно, цикл завершается, когда счетчик становится меньше конечного значения цикла.

Например, если счетчик должен изменяться от 1 до 3 (с шагом по умолчанию равным 1), то цикл исполнится три раза: на первой итерации значение счетчика будет равно единице, на второй оно увеличится на единицу и будет равно двум, на третьей - трем. При попытке начать четвертую итерацию значение счетчика увеличится до четырех, что превышает конечное значение, заданное в заголовке цикла. Поэтому четвертой итерации не будет.

В качестве начального и конечного значения могут выступать как константы, так и выражения. Главное, чтобы все они имели целый тип.

Не следует изменять значение счетчика цикла, используя оператор присваивания. Если сделать это внутри цикла, то может быть нарушена логика последовательного изменения значения счетчика и цикл будет выполнен неправильное число раз.

Приведем примеры целочисленных типов:

```
-- цикл, который будет исполнен три раза:
для Счетчик = 1..3 ЦИКЛ
  -- ...тело цикла
КОНЕЦ;
-- цикл, число итераций которого зависит от значения переменной
-- ЧислоРаботников:
для Счетчик = 1..ЧислоРаботников ЦИКЛ
  -- ...тело цикла
КОНЕЦ;
-- цикл, который будет выполнен 5 раз
-- счетчик будет последовательно принимать значения
-- 10, 8, 6, 4, 2
для Счетчик = 10..1 ШАГ -2 ЦИКЛ
  -- ...тело цикла
КОНЕЦ;
```

Во всех примерах переменная *Счетчик* должна иметь тип **ЦЕЛОЕ**.

Часто невозможно заранее предугадать число итераций цикла, поскольку он должен выполняться до определенного момента, а условия, определяющие окончание цикла, могут изменяться.

Например, требуется выбрать из картотеки сотрудников, чьи фамилии начинаются на букву "А". Поскольку штат предприятия периодически меняется, узнать заранее число таких сотрудников невозможно.

Для решения подобных задач в языке ТБ.Скрипт используется цикл с предусловием, в котором итерации повторяются до тех пор, пока условие, заданное в его заголовке, является истинным.

Заголовок цикла с предусловием выглядит следующим образом:

```
ПОКА <выражение логического типа> ЦИКЛ
или
WHILE <logical expression> DO
```

Заголовок начинается с ключевого слова ПОКА (WHILE). За ним следует условие цикла - логическое выражение, определяющее число повторений. Заканчивается заголовок словом ЦИКЛ (DO), после которого следуют операторы, составляющие тело цикла. Тело цикла заканчивается ключевым словом КОНЕЦ (END) или ЛКИЦ (OD).

Если к моменту начала цикла его условие оказывается ложным, цикл не будет исполнен ни одного раза.

При использовании подобных циклов важно задавать условие таким образом, чтобы рано или поздно оно стало ложным. Иначе цикл не будет прекращен, и программа "зациклится". Чтобы избежать этого, рекомендуется "расширять" условие цикла, учитывая в нем различные граничные ситуации.

Рассмотрим приведенный выше пример с картотекой сотрудников. Теоретически возможен вариант, когда все сотрудники предприятия носят фамилии, начинающиеся на букву "А". В этом случае условие "первая буква фамилии отлична от А" никогда не будет выполнено, что неизбежно приведет к ошибкам цикла. Поэтому данное условие следует уточнить "первая буква фамилии отлична от А и в картотеке есть еще сотрудники". При таком условии цикл остановится, обработав всю картотеку, если в ней нет ни одного сотрудника с фамилией, начинающейся не на "А".

Пример использования цикла с предусловием:

```
ПОКА (ПерваяБуква = 'А') И (ДальшеЕстьЗаписи) ЦИКЛ
-- ...тело цикла
КОНЕЦ;
```

Здесь:

ПерваяБуква - строковая переменная, содержащая первую букву фамилии сотрудника,

ДальшеЕстьЗаписи - переменная логического типа, отмечающая окончание картотеки.

Значения данных переменных нужно обновлять всякий раз, когда из картотеки читается информация о новом сотруднике.

Правильнее для первого сотрудника картотеки сделать это непосредственно перед циклом, а для остальных записей картотеки - последними операторами тела цикла

Отладчик

Студия обладает мощным встроенным языком программирования [ТБ.Скрипт](#), на котором можно эффективно воплощать сложные алгоритмы обработки учетной информации. Однако в процессе создания таких алгоритмов неизбежно возникают программные ошибки, поэтому одной из задач программиста является их своевременное обнаружение, точная диагностика и исправление. Для помощи в решении данных задач и предназначен отладчик [ТБ.Скрипт](#) и [языка типовых операций](#), входящий в состав Студии.

Работе отладчика посвящены следующие темы:

[Назначение и основные функции отладчика](#)

[Переход в режим отладки](#)

[Команды отладчика](#)

[Установка контрольной точки](#)

[Список контрольных точек](#)

[Пошаговое исполнение](#)

[Трассировка вызовов](#)

[Продолжение исполнения процедуры](#)

[Прекращение исполнения](#)

[Стек вызовов процедур и функций](#)

[Добавить переменную](#)

[Просмотр переменных](#)

[Добавление контрольной точки](#)

Диалог предназначен для просмотра и изменения значений переменных в режиме отладки проекта, а также для добавления переменных в список переменных, значения которых постоянно выводятся в окне ["Просмотр переменных"](#). Диалог вызывается с помощью команды [Добавить переменную](#).

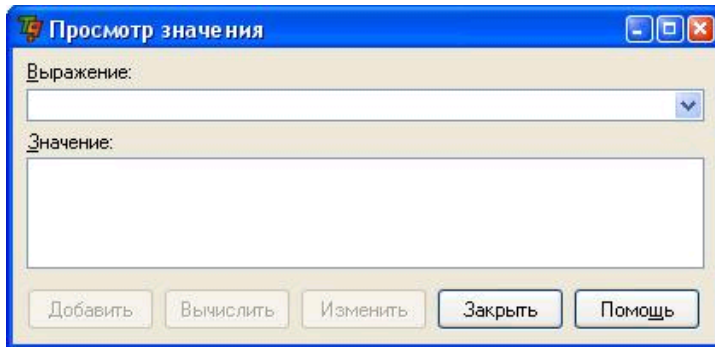


Рис. Добавить переменную.

В поле ввода **Выражение** указывается имя переменной или более сложное выражение, значение которого необходимо контролировать. Если команда была вызвана из контекста редактора текстов в открытом сод-файле, в данное поле автоматически заносится слово, содержащее позицию текстового курсора. Программист имеет возможность отредактировать выражение произвольным образом.

В поле **Значение** отображается текущее значение указанного выражения, однако доступно оно только в том случае, если проект находится в режиме отладки, то есть он запущен на выполнение и приостановлен либо с помощью контрольной точки, либо в результате произошедшей исключительной ситуации. Если проект не находится в режиме отладки, то в поле **Значение** выводится строка "Процесс не запущен". Когда проекта находится в режиме отладки, программист может ввести в данное поле новое значение и нажать кнопку **Изменить**, для того чтобы откорректировать текущее значение выражения (переменной, параметра и пр.).

В момент открытия диалога поле **Значение** сразу заполняется, если в поле **Выражение** было автоматически подставлено правильное по синтаксису выражение (т.е. его можно было вычислить). Если впоследствии программисту необходимо, не закрывая диалога, узнать значение другого выражения, он может ввести его в поле **Выражение** и нажать кнопку **Вычислить**.

По нажатию кнопку **Добавить** указанное выражение добавляется в список, который постоянно отображается в окне ["Просмотр переменных"](#) при отладке.

Кнопка **Закреть** позволяет закрыть диалог.

Назначение диалога: добавление контрольной точки в список.

Информация в поля **Имя класса** и **Номер строки** вводится программой автоматически. В этих полях указывается соответственно имя текущего файла (класса) и номер строки с установленной контрольной точкой.

В поле **Условие** пользователь записывает проверяемое при работе процедуры логическое выражение (например, при использовании циклов). Если условие является истинным, то при достижении данной контрольной точки программа останавливается, в противном случае – точка пропускается. При отсутствии условия контрольная точка считается безусловной, и остановка происходит при первом ее достижении.

Примечание. Если в поле **Условие** для конкретной контрольной точки введено неверное выражение, то при первом ее достижении произойдет остановка программы и будет выдано соответствующее сообщение.

Вызов диалога выполняется командой **Добавить (Ins)** из [списка контрольных точек](#).

Назначение и основные функции отладчика

Отладчик полностью интегрирован во встроенный текстовый редактор Студии. Это позволяет проводить отладку процедур и функций, не меняя привычной среды разработки, и легко переходить от написания текстов к их отладке и обратно.

Отладчик облегчает процесс обнаружения ошибок и позволяет:

- расставлять в тексте программы контрольные точки, в которых исполнение программы будет приостанавливаться;
- исполнять программу в пошаговом режиме - режиме трассировки;
- контролировать последовательность вызовов процедур и функций - стек вызовов;
- просматривать значения переменных в диалоговом окне с возможностью их изменения.

Для использования отладчика требуется предварительно включить режим отладки, установив в [настройках текущего проекта](#) компиляцию с отладочной информацией).

По окончании процесса поиска и исправления ошибок, когда прикладной проект начинает эксплуатироваться в штатном режиме, рекомендуется отключить режим отладки. Это позволит ускорить работу программы на ТБ.Скрипт.

В стандартной поставке инструментальная панель Студии содержит закладку "Проект", на которой располагаются все кнопки, необходимые для проведения отладки. Кроме того, в Главное меню входит аналогичное подменю.

Установка опций отладчика производится на [странице "Отладчик"](#) диалога настройки программы.

Переход в режим отладки

После того как проект скомпилирован с отладочной информацией, его можно запустить на выполнение непосредственно из режима проектирования. При этом существует два варианта приостановления выполнения программы и ее перехода в режим отладки:

1. программа выполняет недопустимое действие (генерирует исключительную ситуацию), в результате чего Студия предлагает разработчику перейти в отладчик;
2. разработчик установил в каком-либо месте исходного кода точку останова (контрольную точку), по достижении которой программа приостанавливается, и Студия переходит в режим отладки.

В режиме отладки на экране открывается окно с исходным текстом программы на ТБ.Скрипт, где характерным цветом (он выбирается пользователем в [окне "Настройки программы"](#)) выделена текущая (исполняемая) строка кода.

Пошаговое исполнение

После того, как исполнение процедуры остановилось на одной из контрольных точек, его можно продолжить в режиме трассировки, т.е. построчного выполнения. С помощью команды [Пошаговое исполнение](#) Студия выполняет все операторы, расположенные на текущей строке отлаживаемой процедуры или функции, и переходит на следующую ближайшую строку, содержащую операторы.

Текущая строка отмечается в редакторе измененным цветом фона (в стандартной настройке цветов - синим).

Если в строке встречается вызов процедуры или функции, то он выполняется как один оператор. Для перехода в вызываемую процедуру или функцию и отладки ее текста следует использовать команду [Трассировка вызовов](#).

Прекращение исполнения

Команда **Прекратить исполнение** позволяет остановить процесс выполнения процедуры или функции и вернуться к режиму редактирования исходного текста или в режим заполнения бланка.

Если проект находится в режиме отладки, на экран выдается запрос о прекращении отладки. При положительном ответе на этот вопрос исполнение процедуры прекращается, при отрицательном - продолжается.

Также прекращение исполнения может быть инициировано в результате изменения исходного текста программы в процессе трассировки. В этом случае, если разработчик подтвердил необходимость продолжать отладку модифицированной программы, сделанные в тексте изменения *игнорируются отладчиком* (фактически он о них ничего не знает, так как работает не с исходным кодом, а с откомпилированным), а положение контрольных точек может уже не соответствовать строкам, где они фактически находятся в откомпилированном представлении COD-файла.

Продолжение исполнения кода

С помощью команды [Продолжить исполнение](#) можно прекратить режим трассировки и продолжить выполнение операторов отлаживаемой процедуры в обычном режиме. Исполнение продолжается либо до первой активной [контрольной точки](#), установленной на одном из исполняемых операторов, либо до конца процесса выполнения проекта.

В процессе трассировки текста кода ТБ.Скрипт программист может воспользоваться возможностью просмотра использующихся в них значений переменных, а также их изменения с целью отладки поведения процедуры. Для выполнения этих действий в многофункциональном окне сообщений Студии предусмотрена страница "Просмотр переменных", активируемая командой [Переменные](#).

В данном окне из всплывающего меню доступны следующие операции:

- добавление переменной в окно просмотра (вызывается также нажатием клавиши Insert);
- удаление переменной из данного окна (*Delete*);
- изменение имени текущей переменной (*Enter*);
- изменение значения текущей переменной (*Enter*).

В данном окне можно просматривать [глобальные и локальные переменные](#) базовых и объектных типов, параметры процедур и функций, а для бланков, являющихся редакторами картотек, - и значения картотечных полей. Информация о каждой переменной включает ее имя, тип данных и текущее значение. В отладочных целях значение переменной может быть изменено вручную.

Изменение переменных бланков, с которыми связана формула, смысла не имеет, т.к. данная переменная все равно затем будет пересчитана в соответствии с этой [формулой](#).

Для добавления переменной непосредственно из текстового редактора можно воспользоваться также командой [Добавить переменную](#), установив курсор на соответствующую переменную в тексте программы.

Если переменная с заданным именем не существует или является локальной переменной процедуры или функции, которая в настоящий момент не исполняется, то в поле **Значение** у такой переменной находится строка "Неизвестная переменная", а поле **Тип** остается пустым. Изменить значение такой переменной нельзя.

Если ни одна из процедур или функций не находится в режиме трассировки, все просматриваемые переменные имеют в поле **Значение** запись "Процесс не запущен". Работа с такими переменными также невозможна.

Существенно упростить просмотр значений переменных позволяет следующая особенность отладчика - если задержать курсор мыши над какой-либо переменной или выражением, на экран выводится всплывающая подсказка со значением этой переменной.

По команде [Контрольные точки](#) на экране открывается страница "Контрольные точки" многофункционального окна сообщений.

В нем приведен список всех контрольных точек, установленных в различных COD-файлах текущего проекта. В данный список включаются как активные контрольные точки, так и те из них, которые установлены на строках, не генерирующих внутреннего кода.

Дополнительно для каждой контрольной точки может быть задано условие, при котором она сработает. Это позволяет более точно описать отлаживаемую ситуацию и избежать слишком частых прерываний работы процедуры. Для указания такого условия достаточно в соответствующей строке (в окне со списком контрольных точек) написать логическое выражение.

Например, если требуется проконтролировать поведение процедуры только в случае нулевого значения переменной **Счетчик**, то можно уточнить контрольную точку условием вида "Счетчик = 0". В этом случае при всех значениях счетчика, отличных от нуля, процедура прерываться не будет.

Для перехода в описание бланка, где установлена контрольная точка, а также для удаления ставших ненужными точек можно воспользоваться всплывающим меню, которое содержит следующие команды:

- **Перейти к** (*Enter*)- команда позволяет открыть соответствующий файл с исходным кодом и установить в нем курсор на указанную контрольную точку;
- **Добавить** (*Ins*) - добавление новой контрольной точки;
- **Удалить** (*Del*) - удаление выделенной контрольной точки;
- **Удалить все** (*Ctrl+Del*) - удаление всех контрольных точек;
- **Выделить все** (*Ctrl+A*) - выделить все контрольные точки;
- **Включена** - включение/отключение выделенной контрольной точки. Когда данный пункт помечен флажком (✓), точка останова включена, иначе – выключена. Отключенная контрольная точка также выделяется особым цветом – по умолчанию это светло-зеленый.
- **Включить все** - включить все контрольные точки, в этом случае все точки выделяются красным цветом;
- **Выключить все** - временно отключение всех контрольных точек из списка;
- **Фиксировать** - зафиксировать или нет окно. Если окно зафиксировано (слева от команды установлен флаг ✓), то нельзя изменить режим открытия окна, т.е. плавающее окно не может стать встраиваемым (прижатым к каким-либо границам). Если окно не является фиксированным (флаг снят), то статус окна может меняться, т.е. встроенное окно может стать плавающим и наоборот;
- **Заккрыть** (*Esc*) - закрывает окно сообщений.

Отладчик программ ТБ.Скрипт способен наглядно представлять последовательность вызова процедур и функций в процессе их исполнения. Эта возможность доступна из окна просмотра стека вызовов, открывающегося на экране по команде [Стек вызовов](#).

Окно просмотра стека вызовов представляет собой страницу в многофункциональном окне сообщений Студии. В нем перечислены процедуры и функции в порядке их вызова при исполнении алгоритма. Самую верхнюю строку списка занимает выполняемая в данный момент процедура, т.е. та, в которой находится текущая исполняемая строка. На самой нижней строке располагается процедура, запущенная первой и инициировавшая все вышележащие вызовы. Сама эта процедура могла быть вызвана пользователем в результате некоторого его действия (то есть в качестве обработчика того или иного [события](#), например, нажатия на кнопку).

Возможен переход из стека вызовов в текст описания соответствующей процедуры по клавише *Enter* или с помощью команд всплывающего меню.

Трассировка вызовов

Команда [Трассировка вызовов](#) позволяет в режиме пошагового исполнения производить отладку текстов процедур и функций, вызываемых из текущей строки программы. В остальном данная команда аналогична команде [Пошаговое исполнение](#).

Например, производится трассировка следующего фрагмента процедуры:

```
Счет = "50";  
СуммаОборота = ВзятьОборот(Счет);  
Сообщение("Оборот="+Стр(СуммаОборота));
```

Текущей является первая строка. Если выполнить команду [Пошаговое исполнение](#), то вызов процедуры **ВзятьОборот** будет произведен как один оператор, и текущей станет третья строка данного фрагмента. Напротив, при исполнении команды **Трассировка вызовов** текущей станет первая строка процедуры **ВзятьОборот**, и можно будет провести пошаговое исполнение ее текста. Когда оно закончится, исполнение вернется в вызвавшую процедуру и текущей строкой станет третья строка приведенного здесь фрагмента.

Если в текущей строке присутствует несколько вызовов, они будут трассироваться по очереди.

Если в вызванной процедуре также есть вызовы других процедур или функций, то можно провести и их трассировку и т.д. Трассировать стандартные процедуры и функции, встроенные в Студию, нельзя.

Использование команд [Пошаговое исполнение](#) и [Трассировка вызовов](#) позволяет визуально проконтролировать работу процедуры, правильность расстановки условных операторов, число повторений циклов и т.д.

С помощью команды [Контрольная точка](#) (*Alt+Del*) в тексте процедуры или функции можно отметить строку с операторами, перед которыми исполнение процедуры будет приостановлено.

Строка, в которой установлена контрольная точка, подсвечивается в текстовом редакторе Студии характерным цветом (по умолчанию, это красный).

Цвет контрольной точки может быть изменен с помощью [диалога настройки параметров редактора](#).

Контрольная точка устанавливается на той строке, на которой в данный момент стоит курсор. На строке, в которой нет ни одного оператора или есть текст, вообще не относящийся ни к одной из процедур бланка, контрольная точка устанавливается в так называемое неактивное состояние, изменяя при этом цвет. Такая контрольная точка действовать не будет и называется запрещенной. Кроме того, некоторые строки программы не генерируют исполняемого кода (например, комментарии). Установленные на таких строках контрольные точки также будут неактивными.

Например, в следующем условном операторе

```
IF R1.ФИО = 'Сидорчук' THEN  
    Message( 'Начальник' );  
ELSE  
    Message( 'Подчиненный' );  
END;
```

третья строка (ELSE) при компиляции не генерирует собственного кода. Поэтому работа установленной на ней контрольной точки невозможна. Однако контрольную точку можно установить на любом из вызовов функции Message или на начало самого условного оператора.

Запрещенные контрольные точки выделяются другим цветом - по умолчанию желтым, который можно изменить.

В режиме отладки в окне редактора слева от строк, имеющих отладочную информацию, отрисовываются кружочки. Это позволяет определить, куда имеет смысл ставить точку останова, а куда - нет.

Установленные контрольные точки сохраняются в Студии от сеанса к сеансу. После загрузки списка контрольных точек проверяется корректность их расстановки и точки, стоящие на строках, не генерирующих внутреннего кода, деактивируются и изменяют свой цвет. Список всех установленных в данный момент контрольных точек выводится на странице "Контрольные точки" многофункционального окна сообщений, которое можно открыть с помощью команды **Сервис|Сообщения**.

Снять контрольную точку можно, повторно выполнив команду **Контрольная точка** для строки, в которой она уже установлена. Кроме этого, контрольная точка (или все точки) может быть временно отключена, а затем снова включена с помощью команд контекстного меню окна со [списком контрольных точек](#).

Если проект приостановлен в режиме отладки, существует дополнительный способ быстрой установки/снятия точки останова - для этого достаточно щелкнуть мышью на кружочке слева от строки кода. При этом в настройках программы на странице ["Отладчик"](#) включен флаг **Показывать строки с кодом**.

Препроцессор - это специальный механизм предварительной обработки исходного текста на ТБ.Скрипт, позволяющий изменять некоторые аспекты его компиляции. Для управления препроцессором существуют определенные синтаксические конструкции - директивы. Они выполняются компилятором непосредственно в процессе разбора исходного кода.

Препроцессор не меняет самого исходного текста, однако может в соответствии с директивами либо пропускать его участки, исключая тем самым из откомпилированного двоичного представления программы, либо наоборот включать их туда. Такая компиляция называется условной, поскольку происходит не всегда, а при выполнении некоторого условия, указанного с помощью директивы.

В описании классов на языке ТБ.Скрипт используется директива условной компиляции `#If`.

С ней логически тесно связана другая директива `#Define`, предназначенная для определения так называемых *ключей компиляции* - произвольных поименованных констант, обозначающих условия (или варианты) компиляции. Например, с помощью ключа компиляции можно задать номер рабочей версии класса и в зависимости от него решать (с помощью директивы `#If`) использовать или нет некоторые свойства этого класса.

Также с помощью препроцессора можно генерировать предупреждения и сообщения об ошибках, по аналогии с тем, как это делает компилятор. Для этих целей используются директивы `#Предупреждение` и `#Ошибка`.

Директивы препроцессора описываются в следующих параграфах:

[Директива #Определить / #Define](#)

[Директива #Если / #If](#)

[Директива #Предупреждение / #Warning](#)

[Директива #БезПредупр / #NoWarning](#)

[Директива #Ошибка / #Error](#)

[Пример использования директив условной компиляции](#)

Данная директива позволяет подавить вывод нескольких следующих предупреждений компилятора. Директива имеет приоритет перед настройками, сделанными в диалоге ["Настройка сообщений"](#). То есть, даже если в настройках программы вывод предупреждений включен, директива, встреченная в исходном коде, временно превалирует над этими настройками и запрещает вывод предупреждений. Директива влияет только на предупреждения, генерируемые компилятором, и не подавляет предупреждений, в явном виде вызываемых самим разработчиком с помощью директивы #Предупреждение.

Синтаксис:

```
#БезПредупр [<целое положительное число>];  
#NoWarning [<integer positive number>];
```

Здесь <целое положительное число> - это необязательный аргумент, определяющий, сколько следующих предупреждений будет подавлено. Если аргумент опущен, то по умолчанию подавляется одно предупреждение.

Директива действует только в том модуле, где она вставлена. Иными словами, действие одной директивы не переносится на другие модули, компилирующиеся после текущего, даже если указанное количество подавляемых предупреждений не исчерпано.

Данная директива формирует блок условной компиляции. Ее синтаксис:

```
#Если <Условие> Тогда
...
[#Еслиже]
...
[#Иначе]
...
#Конец
```

или

```
#If <Condition> Then
...
[#Elsif]
...
[#Else]
...
#End
```

В качестве условия выступает выражение логического типа, в котором можно использовать константы и ключи компиляции, определенные ранее в данном классе или глобально в проекте. Если условие истинно, то часть текста, заключенная в первом блоке (между директивой **#Если** и следующей директивой **#Еслиже**, **#Иначе** или **#Конец**), разбирается компилятором; если ложно, то данный блок пропускается, как один большой комментарий и выполняется другой блок (начиная с ключевого слова **#Еслиже**), условие которого истинно. Блоков **#Еслиже** может быть произвольное количество. Если ни одно из условий не срабатывает, компилятор обрабатывает оставшийся блок между директивами **#Иначе** и **#Конец**, если он есть. Кстати, конструкцию

```
#If False then
...
#End
```

можно использовать для комментирования больших кусков текста.

Блоки условной компиляции могут быть вложенными.

Синтаксис данной директивы следующий:

```
#Определить <ИмяКлючаКомпиляции> = <Значение> ;  
#Define <CompilationKeyName> = <Value> ;
```

Директива определяет ключ компиляции - переменную, используемую далее в условиях компиляции. В качестве значения можно записать формулу, содержащую операции над константами и другими ключами компиляции, определенными ранее в данном классе или глобально для проекта.

В Студии определены три встроенных (глобальных) ключа компиляции, доступные в любом классе проекта:

- **Версия|Version**, содержащий номер версии Студии (например, '7.0');
- **ТекущийПроект|CurrentProject**, содержащий имя текущего компилируемого проекта;
- **Отладка|Debug**, логическое значение, равное истине, если проект компилируется с отладочной информацией, или лжи в противном случае.

Глобальные для проекта ключи можно задавать в диалоге [настройки параметров проекта](#).

Значение любого ключа компиляции, кроме вышеупомянутых встроенных, можно переопределить другой директивой #Define, в том числе и задав ключу значение другого типа.

Следует иметь в виду, что ключи компиляции не пересекают границ проектов. Иными словами, ключи подпроекта не известны в использующем его проекте.

Данная директива порождает ошибку компиляции с заданным текстом. Выдачу такой ошибки подавить нельзя (в отличие от предупреждения).

Синтаксис:

```
#Ошибка <текст> ;  
#Error <текст> ;
```

Здесь <текст> - строковый литерал (строка в кавычках или апострофах).

Данная директива может использоваться в процессе разработки систем на ТБ.Скрипте для пометки мест, без реализации которых система не имеет смысла.

Например, в разрабатываемой системе есть ряд процедур, без реализации которых нельзя выпустить окончательный вариант продукта. Они пока не реализованы, но о них важно не забыть. Как это сделать?

1) заводим глобальный ключ компиляции логического типа, например, с именем AllDone и присваиваем ему False;

2) во всех критичных местах пишем приблизительно такой код:

```
proc DoSomeone;  
  -- процедура пока не реализована  
  #Warning "Не забыть реализовать процедуру DoSomeone!";  
  #If AllDone Then  
    #Error "Нельзя выпускать продукт без данной процедуры!!!";  
  #End  
end;
```

В результате при каждой перекомпиляции данного класса компилятор будет напоминать о необходимости вернуться к данной процедуре. Если же Вы установите ключу AllDone значение True (т.е. Вы считаете, что система полностью реализована) и сделаете перекомпиляцию проекта, то компилятор возбудит на этом месте ошибку и не допустит создания незавершенного проекта.

Директива `#Предупреждение` / `#Warning`

Данная директива задает строку, которая будет выводиться в качестве предупреждающего сообщения в процессе компиляции описания класса. Вывод данного сообщения можно подавить, отключив соответствующий флаг в диалоге [настройка сообщений](#).

Синтаксис:

```
#Предупреждение <текст> ;  
#Warning <текст> ;
```

Здесь <текст> - строковый литерал (строка в кавычках или апострофах).

Данная директива может использоваться в процессе разработки систем на ТБ.Скрипте для пометки мест, к которым нужно не забыть вернуться.

Пример:

```
#Define Условие = 0; -- задаем ключ компиляции с именем "Условие"
#Define Условие = 1; -- можем переопределить его значение

#If Отладка then
  -- все это будет работать только при компиляции отладочной версии проекта
  #If Условие > 0 then
    -- эта часть текста будет компилироваться (т.к. Условие > 0)
    #If False then
      -- эта часть текста не будет компилироваться никогда
    #Else
      -- эта часть текста будет компилироваться
    #End
  #Else
    -- эта часть текста не будет компилироваться (т.к. Условие > 0)
    #If False then
      -- эта часть текста не будет компилироваться никогда
    #Else
      -- эта часть текста могла бы компилироваться, если бы Условие <= 0
    #End
  #End
#End
```



Разрабатываемые в программе проекты строятся на принципах объектно-ориентированного программирования. Иными словами, практически все сущности программы (функции, пользовательский и программный интерфейсы) представлены в виде объектов различных классов, задающих особенности и механизмы их функционирования.

Для просмотра иерархии классов используется программное средство - браузер иерархии классов, вызываемый командой [Проект|Иерархия классов](#). С помощью браузера *разработчик может оперативно получить краткую справку о классах, объектных конструкциях языка ТБ.Скрипт, а также осуществить быстрый поиск и навигацию по иерархии.*

Окно "Иерархия классов" разделено на две части. В левой его половине в иерархическом виде выводятся все зарегистрированные в системе классы как встроенные, так и пользовательские классы, введенные разработчиком проекта. Пользовательские классы отображаются в иерархии только после компиляции проекта. Напомним, что и бланки, и картотеки, и записи (документы) представляют собой классы языка ТБ.Скрипт.

Правая половина окна "Иерархия классов" содержит две страницы "[Свойства](#)" и "[Состав](#)". Они используются для отображения более детальной информации о параметрах, методах и свойствах того класса, который в данный момент выделен в иерархии. Переключение между страницами осуществляется с помощью закладок, расположенных в нижней части окна.

Часть классов, отображаемых в левой части окна, является встроенной, так как реализуется внутри самой системы и может быть изменена только программистами фирмы, разрабатывающей программу. К встроенным классам относятся классы верхнего уровня "Объект", "Система", "Консоль" и "Бухгалтерия", которые выступают в качестве родительских для всех остальных классов языка ТБ.Скрипт.

Встроенные классы помечаются в иерархии с помощью значка желтой шестеренки . Чтобы отличить пользовательские классы, определенные прикладным разработчиком, от встроенных классов, они помечаются шестеренкой другого цвета .

В окне "Иерархия классов" доступны следующие команды контекстного меню:

- [Свойства класса](#)
- [Свойства объекта](#)
- [Свойства предков](#)
- [Ссылки](#)

Назначение: включает/отключает режим отображения [в окне с иерархией классов](#) тех свойств, который являются [свойствами класса](#) (в отличие от свойств объекта - экземпляра класса).

Вызов команды в стандартной настройке интерфейса осуществляется с помощью с помощью контекстного меню [в окне с иерархией классов](#).

Для [пользовательских классов](#), реализованных на языке ТБ.Скрипт, свойства, относящиеся к конкретному объекту, задаются с помощью директивы [InClass](#).

Для встроенных классов Студии *все свойства классов* определены на внутреннем системном уровне.

Назначение команды: включает/отключает режим отображения [в окне с иерархией классов](#) тех свойств, который являются [свойствами объектов](#) (в отличие от свойств класса, общих для всего класса).

Вызов команды в стандартной настройке интерфейса осуществляется с помощью с помощью контекстного меню [в окне с иерархией классов](#).

Для [пользовательских классов](#), реализованных на языке ТБ.Скрипт, свойства, относящиеся к конкретному объекту, задаются с помощью директивы [InObject](#).

Для встроенных классов Студии все *свойства объектов* определены на внутреннем системном уровне.

Назначение команды: включает/отключает режим отображения [в окне с иерархией классов](#) тех свойств, которые наследуются [от родительских классов](#).

Наследуемые свойства отображаются отличительным цветом (в стандартной цветовой палитре они - серые, в то время как собственные свойства - черные).

Вызов команды в стандартной настройке интерфейса осуществляется с помощью с помощью контекстного меню [в окне с иерархией классов](#).

На странице "Свойства" [окна "Иерархия классов"](#) выводятся синонимы класса, название пакета (или проекта), содержащего класс, тип класса и характеристики вызова динамических свойств. Рассмотрим эти атрибуты более подробно.

Синонимы класса - это альтернативные имена классов, которые могут использоваться наравне с основным именем, отображенным в иерархии. Дополнительно синонимы всегда отображаются в строке состояния в самом низу главного окна Студии, что весьма полезно, когда в браузере открыта страница ["Состав"](#), а не "Свойства".

Пакет - это независимый набор логически связанных программных модулей, образующий в иерархии своего рода надкласс верхнего уровня (в терминах программирования он определяет самостоятельное пространство имен). Пакеты могут состоять как из встроенных классов, так и из классов, определенных прикладным программистом с помощью языка ТБ.Скрипт или MTL. Так, все встроенные классы описаны в пакете Ядро. А каждый прикладной проект образует самостоятельный пакет, причем если проект использует подпроекты, то в его составе будет несколько пакетов. Как правило, имя пакета необходимо для указания полностью квалифицированного имени свойства (переменной или метода) какого-либо класса при доступе из другого пакета (например, ПакетА.Класс1.МетодИкс). Однако для упрощения программирования Студия позволяет опускать имя пакета Ядро и даже, более того, имена встроенных классов (Система, Консоль, Бухгалтерия).

Тип класса может принимать одно из двух значений: встроенный класс и пользовательский класс. Их смысл описан выше.

Последняя характеристика, отображаемая на странице свойств класса - **динамический вызов свойств**. Он может поддерживаться или не поддерживаться классом. Динамический вызов свойств представляет собой особый механизм генерации кода программы на ТБ.Скрипт, при котором допускается использовать обращение к свойствам классов-потомков по отношению к переменным родительских классов. В теории программирования эта технология известна под названием "позднее связывание". Суть ее в том, что в программе часто бывает удобно использовать переменную базового класса для хранения значений, которые фактически относятся к производным классам. Например, в системе существует класс "Запись" ("Документ"), и на его базе созданы производные классы "Счет", "Оплата", "Отгрузка". Тогда можно воспользоваться массивом объектов типа "Запись" для перебора документов всех производных типов. Внутренняя сложность заключается здесь в том, что в производных классах могут быть определены свойства, которые отсутствуют в базовом классе и не известны на стадии компиляции соответствующих строк программы на ТБ.Скрипт. Именно в таких случаях и применяется динамический вызов свойств, который фактически заключается в том, что компилятор откладывает проверку валидности вызова свойства до стадии исполнения программы.

Таким образом, динамический вызов свойств делает программу более гибкой, но может привести к ошибкам, не выявленным на стадии компиляции. Кроме того, он производится медленнее, чем статический вызов свойств. Динамический вызов свойств позволяет реализовать одну из важнейших особенностей объектно-ориентированного программирования - полиморфизм.

На странице "Состав" [окна "Иерархия классов"](#) выводится список свойств (полей, процедур, функций, констант, пользовательских типов) текущего класса. Список имеет две колонки: "Тип" и "Формат".

Первая колонка "Тип" позволяет определить, что же именно представляет из себя свойство - поле, константу, процедуру, функцию, событие или пользовательский тип (см. далее).

Поле представляет собой [переменную](#) класса или объекта, или же [поле записи](#), если речь идет о классе, производном от класса **Запись**. [Константы](#) - это особый случай неизменяемых полей. [Процедуры и функции](#) - методы классов и объекты. [Пользовательские типы](#) описываются в исходном тексте ТБ.Скрипт с помощью ключевого слова **тип (type)**. Если пользовательский тип является перечисляемым, то его представление в браузере классов представляет собой не просто строку, а составной элемент, который можно как бы раскрыть, нажав на значок '+' (плюс) слева от слова "тип". В результате появятся значения, определенные в данном типе.

Каждый элемент класса, в соответствии со своим смыслом, обозначается определенным значком, так что, например, поле, доступное только на чтение, визуально легко отличить от поля, для которого разрешены и чтение, и запись.

Вторая колонка "Формат" содержит формализованную запись синтаксиса свойства, включая его имя, тип данных (например, строка, целое, запись), список аргументов для процедур и функций. Например, поля описываются следующим образом:

```
поле <ИмяПоля> : <ТипПоля> ;
```

Описание процедуры выглядит так:

```
проц <ИмяПроцедуры> ( <ИмяАргумента> : <ТипАргумента>  
[ , <ИмяАргумента> : <ТипАргумента> ] ) ;
```

Список аргументов состоит из последовательности пар "<ИмяАргумента>: <ТипАргумента>", разделенных запятыми. Вся последовательность заключена в круглые скобки, причем она может отсутствовать, если аргументов нет. В этом случае круглые скобки также не выводятся.

Функция отличается от процедуры лишь тем, что возвращает значение определенного типа:

```
функ <ИмяФункции> ( <ИмяАргумента> : <ТипАргумента>  
[ , <ИмяАргумента> : <ТипАргумента> ] ) : <ТипВозвращаемогоЗначения> ;
```

Необязательные параметры, которые могут быть опущены при вызове процедуры или функции, отображаются в квадратных скобках. В простейших случаях, когда у процедуры или функции нет ни одного аргумента, запись существенно упрощается:

```
функ <ИмяФункции> : <ТипВозвращаемогоЗначения> ;
```

Особый случай представляет собой свойство-событие. На уровне объекта событие представляет собой поле строкового типа, в котором хранится имя процедуры или функции - обработчика [события](#) (при этом соответствующий метод, разумеется, должен быть описан в классе). Однако в браузере классов событие представлено не как поле, а как описание прототипа обработчика. Таким образом, разработчик легко может узнать синтаксис требуемого метода. Напомним, что обработчик события, как и ряд других свойств, может быть назначен для интерфейсных классов объектов с помощью [визуального редактора шаблонов бланков](#).

В верхней части окна "Состав" перечисляются свойства непосредственно класса, то есть статические свойства, доступ к которым обеспечивается при указании имени класса (например, Изображение.Создать, где Изображение - это имя встроенного класса, а Создать - имя функции). Статическое свойство существует в единственном числе для всего класса и может быть использовано, даже если еще не создано ни одного элемента данного класса.

Далее в окне перечисляются свойства объектов класса, то есть динамические свойства, описывающие конкретные экземпляры объектов. Доступ к таким свойствам осуществляется через имя переменной соответствующего типа (например, И.ЗагрузитьИзФайла(Ф), где И - это проинициализированная переменная класса Изображение, а ЗагрузитьИзФайла - имя процедуры объекта). Динамические свойства у каждого объекта свои и могут быть использованы только в течении времени жизни соответствующей переменной данного класса.

Пользователь может выборочно включать/отключать отображение статических и динамических свойств. Для этой цели предназначены команды контекстного меню - "Свойства класса" и "Свойства объекта". Когда рядом с соответствующим пунктом стоит метка, такие свойства отображаются в списке.

Кроме того, с помощью контекстного меню возможно включать/отключать режим отображения в списке тех свойств, которые наследуются от родительских классов. Такие свойства отображаются серым цветом.

По умолчанию свойства перечислены в порядке их описания внутри классов, однако пользователь может

отсортировать их по алфавиту, нажав на заголовок колонки Формат.

По названиям свойств можно проводить контекстный поиск (по подстрокам) - достаточно воспользоваться командами **Поиск и Повтор**. Например, если установить курсор на начало иерархии, вызвать команду **Поиск** и ввести в открывшемся диалоге текст для поиска "транзак", то браузер найдет первое свойство, в название которого входит указанная строка (это будет процедура **ЗавершитьТранзакцию**), и затем будет последовательно переходить на другие свойства с данной строкой в названии, если пользователь продолжит поиск с помощью команды **Повтор**.

Пользователь может просмотреть все места исходного кода, где упоминается какое-либо свойство (поле, константа, метод, тип, событие), дважды щелкнув на нем мышью или выполнив команду **Ссылки** контекстного меню. В результате открывается окно "Ссылки", в котором перечислен список с указанием имен файлов с исходным текстом и номеров строк в них, где используется это свойство или класс. Для перехода к нужной строке исходного текста следует дважды щелкнуть на нужной строке списка или воспользоваться командой **Открыть файл** (*Enter*) контекстного меню.

Директивы языка ТБ.Скрипт изложены в следующих темах:

[Класс / Class](#)

[Бланк / Blank](#)

[ВОбъекте / InObject](#)

[ВКласе / InClass](#)

[Импорт / Import](#)

[Публично / Public](#)

[Лично / Private](#)

Директива **Бланк/Blank** открывает описание класса бланка - совокупности организованного в процедуры и функции кода, а также переменных, используемых бланком. Бланк является классом особого вида, поэтому допускается описание бланков с помощью директивы Class. Формат директивы **Blank** следующий:

```
Blank [Inherited <BaseClassName>]
    "Name"
    [ , Editor <RecordClass>]
    [Синоним <AuxClassName_0> [{ ,<AuxClassName_i> }]];
Бланк [Унаследован <ИмяБазовогоКласса>>]
    "Имя"
    [ , Редактор <КлассЗаписи>>]
    [Синоним <ДопИмяКласса_0>> [{ ,<ДопИмяКласса_i> }]];
```

После ключевого слова **Blank** может следовать необязательный блок, начинающийся с ключевого слова **Inherited**. Данный блок используется для указания наследования описываемым бланком всех свойств бланка BaseClassName. При этом описываемый бланк становится производным от BaseClassName. Если описываемый бланк является редактором записей (документов) определенного типа, то после имени бланка через запятую следует ключевое слово **Редактор/Editor** и название типа записей (документов).

Как и в случае простого класса в заголовке бланка при необходимости можно дополнительно указать один или несколько [синонимов](#).

Бланк должен быть описан единым блоком, который начинается с вышеуказанной директивы **Blank** и заканчивается ключевым словом Конец (**End**). Весь код бланка, включая процедуры, функции и переменные, должен быть целиком описан в одном модуле с расширением COD. Кроме того, внешний вид бланка и его интерфейсные элементы определяются шаблоном бланка, который хранится в файле с расширением TPL, имеющем то же имя, что и COD-файл.

Директива **InClass** задает начало фрагмента кода, в котором описываются статические свойства класса, то есть процедуры, функции и переменные (поля), которые определяют свойства непосредственно класса, а не его объектов. Директива имеет следующий формат:

```
InClass
```

К статическим свойствам можно обращаться по полному имени, включающему имя класса и, через точку, имя свойства. Статические свойства доступны всегда, вне зависимости от того, создан ли хоть один объект данного класса.

Фрагмент кода со статическими свойствами начинается непосредственно после директивы **InClass** и продолжается до следующей директивы [InObject](#) или конца модуля с исходным кодом.

Директива **InObject** задает начало фрагмента кода, в котором описываются динамические свойства класса, то есть процедуры, функции и переменные (поля), к которым можно обращаться только из предварительно созданных объектов данного класса. Иными словами, свойства из раздела InObject представляют собой свойства конкретных экземпляров класса, и каждый объект имеет индивидуальный набор значений данных свойств. Директива имеет следующий формат:

InObject

Напомним, что при адресации к динамическим свойствам в коде полное имя свойства включает имя объекта и, после точки, имя свойства.

Фрагмент кода с динамическими свойствами начинается непосредственно после директивы **InObject** и продолжается до следующей директивы [InClass](#) или конца модуля с исходным кодом. По умолчанию, то есть если явно не указано обратное, код модуля считается динамическим. Например, свойства описанные сразу после директивы Class являются динамическими (описывают экземпляры класса).

Директива **Import** предназначена для упрощения записи полностью квалифицированных (составных) имен свойств других классов, в том числе, определенных в других проектах (подпроектах) и классах **Система**, **Бухгалтерия**, **Консоль**. Например, если в проекте (пакете) Тест1 есть Класс1 со статическим свойством Свойство11, то для вызова Свойства11 из кода другого проекта (пакета) необходимо записать строку вида:

```
Тест1.Класс1.Свойство11
```

Если в коде много таких обращений к "внешним" классам, то программа становится громоздкой. Директива **Import** позволяет решить эту проблему. Достаточно один раз в начале кода класса описать с помощью директивы внешние классы и затем обращаться к свойствам импортируемых классов по кратким названиям.

Синтаксис директивы следующий:

```
Import [<ИмяВнешнегоПакета>] Classes <ИмяКласса> [ , <ИмяКласса>...];
```

или

```
Импорт [<ИмяВнешнегоПакета>] Классы <ИмяКласса> [ , <ИмяКласса>...];
```

Например, записав в начале некоторого класса НовыйКласс директиву:

```
Import Тест1 Classes Класс1, Класс2, Класс3;
```

программист может затем обращаться к свойствам классов Класс1, Класс2 и Класс3 напрямую, в том числе, написать просто:

```
Свойство11;
```

Если имя пакета (проекта) опущено, то импортируемые классы по умолчанию ищутся в текущем проекте.

Директива **Класс/Class** открывает описание класса - совокупности организованного в процедуры и функции кода, а также переменных. Формат директивы следующий:

```
Class [Inherited <BaseClassName>]
  "Name"
  [, Editor <RecordClass>]
  [Synonym <AuxClassName_0> [{,<AuxClassName_i>}]];
Класс [Унаследован <ИмяБазовогоКласса>]
  "Имя"
  [, Редактор <КлассЗаписи>]
  [Синоним <ДопИмяКласса_0> [{,<ДопИмяКласса_i>}]];
```

После ключевого слова **Class** может следовать необязательный блок, начинающийся с ключевого слова **Inherited**. Данный блок используется для указания наследования всех свойств класса BaseClassName в описываемом классе, который становится производным от BaseClassName. Если описываемый класс представляет собой код бланка-редактора, то после имени класса через запятую следует ключевое слово **Editor** и название типа записей (документов), которые будут редактироваться с помощью данного бланка.

В случае необходимости в заголовке можно дополнительно указать один или несколько синонимов для имени класса. Синонимы задаются в конце директивы **Класс (Class)**. Список синонимов начинается с ключевого слова Синоним (**Synonym**)**Synonym**, после которого следуют разделенные запятыми идентификаторы. Синонимы могут использоваться в программе на ТБ.Скрипт точно также, как и основное имя класса, что иллюстрируется развернутым [примером](#).

Класс должен быть описан единым блоком, который начинается с вышеуказанной директивы **Класс** и заканчивается ключевым словом **Конец (End)**. Весь класс, включая процедуры, функции и переменные, должен быть целиком описан в одном модуле с расширением COD.

Директива Лично / Private

Директива **Private** задает начало фрагмента кода, в котором описываются свойства и методы класса, закрытые для доступа извне. Обращаться к личному свойству или вызвать личный метод можно только непосредственно из данного класса или из производных классов.

Директива имеет следующий формат:

```
Private
```

Фрагмент кода с личными свойствами начинается непосредственно после директивы **Private** и продолжается до следующей директивы [Public](#) или до конца модуля с исходным кодом

Директива **Public** задает начало фрагмента кода, в котором описываются свойства и методы класса, открытые для доступа извне. Таким образом, из любого места программы на языке ТБ.Скрипт можно обратиться к публичному свойству или вызвать публичный метод.

Директива имеет следующий формат:

```
Public
```

Фрагмент кода с публичными свойствами начинается непосредственно после директивы **Public** и продолжается до следующей директивы [Private](#) или до конца модуля с исходным кодом.

Из внешних классов можно обращаться к публичным свойствам и методам по квалифицированному имени (с указанием полного имени класса) или же по краткому имени, если в начале модуля стоит директива [Import](#).

```
-- файл A.COD
class "Первый" synonym First;
InClass public
X Synonym Икс : Numeric = 10.0;
  proc Test1;
    Trace(ClassName + ':Hello!');
    Second.Test1; -- эквивалентно вызову B.Test1
  end;
  -- Test2 принимает объект класса B
  proc Test2 (Obj : Second);
    Obj.Test1; -- эквивалентно Obj.Test1
  end;
end

-- файл B.COD
class "Второй" synonym Second;
InClass public
  proc Test1 synonym Тест1;
    Trace(ClassName + ':Hello!');
  end;
  proc Test2 synonym Тест2;
    -- выводим имя класса A
    Trace(First.ClassName);
    -- выводим значение переменной A.X
    Trace(First.Икс);
  end;
end
```

Поскольку встроенный в Студию язык программирования ТБ.Скрипт относится к числу объектно-ориентированных, описание его конструкций и приемов работы с ним необходимо предварить общим изложением основ объектно-ориентированного подхода (ООП).

Двумя основополагающими понятиями ООП являются **класс** и **объект**. *Класс - абстрактная программная сущность, описывающая некоторую структуру данных и способы работы с ней.*

То, как именно описывается класс, зависит от конкретного средства разработки и не играет особой роли в идеологии ООП. Здесь мы ограничимся рассмотрением лишь наиболее общих принципов, а конкретные сведения по программированию классов в Студии будут приведены в следующих темах:

[Понятие класса](#)

[Модули классов](#)

[Регистрация классов в проекте](#)

[Описание класса](#)

[Заголовок класса](#)

[Наследование и полиморфизм](#)

[Свойства класса и объектов](#)

В классе могут быть определены поля (или переменные) различных типов, хранящие информацию, а также процедуры и функции, оперирующие значениями этих полей. Процедуры и функции часто называются обобщающим термином методы. В свою очередь, всё, что входит в описание класса и фактически отличает один класс от другого (то есть и поля, и методы), будет называться в рамках данного материала свойствами класса.

Класс абстрактен в силу того, что он суть только описание структуры данных и способов работы с ней. Класс не есть сама структура данных.

На основе описания класса можно создавать *объект данного класса*, то есть переменную особого типа. Именно этот объект содержит конкретные значения полей, определенных в классе, и именно над объектом производятся манипуляции с помощью алгоритмов, заложенных в классе. Говорят, что объект - это конкретный экземпляр класса. Для каждого класса может быть создано произвольное число объектов, причем каждый из них позволяет хранить собственный набор значений полей. Получается, что объекты одного класса идентичны друг другу с точки зрения их внутренней структуры и методов работы с ними, но при этом могут хранить различные значения.

Для пояснения вышеизложенного приведем пример. Пусть определен класс **Счет**, в котором имеются поля для хранения номера и даты счета, название контрагента, таблица с товарами. Все это описано один раз - при определении класса. На основе этого определения можно создавать экземпляры счетов, и каждый счет будет содержать информацию (дату, номер, контрагент, товары) из конкретного документа.

Забегая вперед скажем, что в принципе существует возможность определять свойства как относящиеся к классу в целом или к объекту. Первые из них называются статическими, потому что ими можно пользоваться всегда, вне зависимости от того, существует уже хоть один объект класса или нет. Статические свойства существуют в единственном числе для всего класса.

Те свойства, что относятся к объектам, называются динамическими - ими можно пользоваться только с указанием конкретного экземпляра класса (объекта). То есть обращение к динамическому свойству требует наличия объекта. Динамические свойства существуют для каждого объекта свои.

Кроме того, ООП, как правило, допускает разделение свойств класса на личные и публичные. Публичные свойства класса доступны из других классов, в то время как личные - нет. Это позволяет более полно контролировать использование классов и ограничить число программных ошибок.

ТБ.Скрипт предоставляет механизмы для определения статических и динамических, а также личных и публичных свойств классов. Этот вопрос рассматривается в разделе [Директивы языка ТБ.Скрипт](#).

Синтаксис стандартного заголовка класса следующий:

```
Класс "< ПолноеНазвание"[, редактор <ИмяКлассаРедактируемойЗаписи >];  
Class "< FullName >"[, editor < RecordClassToEdit >];
```

В простейшем случае заголовок имеет вид:

```
Класс "<ПолноеНазвание>";
```

Заголовок класса начинается с ключевого слова **Класс** (англоязычный синоним **Class**). Рекомендуется записывать его с первой позиции строки. После этого должно стоять полное название (описание) класса, заключенное в кавычки. Затем может следовать атрибут бланка-редактора. В конце заголовка должна стоять точка с запятой.

В принципе, при описании класса, реализующего бланк (то есть имеющего не только алгоритмическое представление в виде COD-файла, но и экранную форму - TPL-шаблон), можно использовать и ключевое слово **Бланк (Blank)**. При этом будет описан не просто класс, а именно класс бланка. Однако бланки являются подмножеством общего понятия "класс", поэтому в Студии по умолчанию всегда используется более общее описание с ключевым словом **Класс**. Если система обнаруживает в том же каталоге, где находится COD-файл, одноименный TPL-файл, то из этого автоматически следует, что описываемый класс является бланком. Описание бланков приводится в Главе Подсистема бланков.

Еще раз напомним, что если класс описывает бланк-редактор, то заголовок дополняется ключевым словом **Редактор (Editor)** и названием редактируемой записи (структуры данных).

Примеры простых заголовков:

```
Класс "Платежное поручение";  
Class "Работник";
```

Пример класса бланка-редактора:

```
Класс "Накладная", редактор Справочники.Накладные;
```

Строка с названием класса отображается в окне проекта и облегчает прикладным разработчикам понимание сути выполняемых классом функций.

Как правило, название содержит название операции или документа с точки зрения прикладной области автоматизации, например, бухгалтерии. Так, для класса ПОРУЧП - это "Платежное поручение".

В случае необходимости в заголовке можно дополнительно указать один или несколько синонимов для имени класса. Поскольку имя класса соответствует имени COD-файла, в котором класс описан, иногда бывает полезно ввести синоним: например, если файл назван по-английски, имеет смысл описать русскоязычный синоним.

Синонимы задаются в конце директивы **Класс (Class)**. Список синонимов начинается с ключевого слова **Синоним (Synonym)**, после которого следуют разделенные запятыми идентификаторы. Таким образом, синтаксис полного заголовка класса, с учетом синонимов, следующий:

```
Класс "<ПолноеНазвание>"[, редактор <ИмяКлассаРедактируемойЗаписи>]  
[СИНОНИМ <Идентификатор_1> [{, Идентификатор_i}]];
```

Пример:

```
-- файл Накладная.cod; основное имя класса - Накладная  
Класс "Базовое описание накладной" Синоним Накл, Waybill;
```

Здесь в дополнение к имени класса "Накладная" вводятся два синонима: "Накл" и "Waybill".

Вот еще один пример:

```
-- файл Заказ.cod; основное имя класса - Заказ  
Класс "Бланк заказа товаров", редактор Реестр.Заказы Synonym Invoice;
```

Здесь синоним объявляется для класса, который описывает бланк-редактор.

Синонимы могут использоваться в программе на ТБ.Скрипт точно также, как и основное имя класса, что иллюстрируется развернутым примером.

Итак, каждый класс определяется с помощью текстового описания на языке ТБ.Скрипт, расположенного в отдельном COD-файле. Название этого файла и есть имя класса - его идентификатор. Например, модуль ПЛАТЕЖКА.COD описывает класс ПЛАТЕЖКА.

Имя класса используется при обращении к переменным, процедурам и функциям данного класса из других модулей. Именно под этим именем класс регистрируется в иерархии классов Студии и виден в окне иерархии классов.

В принципе, идентификатор может быть составным (квалифицированным), т.е. состоять из нескольких простых идентификаторов, разделенных точками. В этом случае собственно именем класса считается последний простой идентификатор в цепочке, а предыдущие идентификаторы описывают его принадлежность к той или иной логической группе классов. Например, класс с названием "БАНК.Поручп" относится к группе БАНК (банковские документы).

Поскольку в системе Windows имя файла может быть длиной до 255 символов, содержать русские буквы и специальные символы, такие, как точка, то вполне возможным становится задание в качестве имени файла таких конструкций, как "БАНК.ПОРУЧП.COD" или "КАССДОК.ПРИХОРЕД.COD". Соответственно, эти классы будут доступны в прикладных проектах Студии под именами "БАНК.ПОРУЧП" или "КАССДОК.ПРИХОРЕД".

Важно отметить, что группу классов образуют и собранные в одном каталоге модули. Таким образом, если модуль с именем "ПОРУЧП.COD" помещен в каталог "БАНК", то в проекте данный класс также будет иметь имя "БАНК.ПОРУЧП".

Следует иметь в виду, что Студия имеет одно зарезервированное имя класса **Profile**. Если программист реализует такой класс (файл Profile.cod), а в нем - метод **Init**, то этот метод будет вызываться системой автоматически в самом начале загрузки проекта, причем даже раньше, чем процедуры автозапуска, указанные в схеме доступа. Более того, если процедуры автозапуска вызываются только для текущей (выбранной перед запуском проекта) схемы доступа, то **Profile.Init** вызывается для всех проектов, которые участвовали в создании информационной базы, включая и все их подпроекты. Состояние рабочего стола программы, то есть набор и положение всех дочерних окон, сохраненные с предыдущей сессии, считывается до вызова метода **Profile.Init**, что гарантирует инициализацию всех списков истории при появлении каких-либо бланков.

Также в этом модуле **Profile** можно описать специальную процедуру GetSessionInfo, которая предоставляет разработчику возможность ввести дополнительную информацию о текущем проекте во время сессии с помощью вызова процедуры AddSessionInfo.

Одно из неоспоримых преимуществ объектно-ориентированного программирования (ООП) заключается в доступности механизма наследования. **Наследование** позволяет создавать классы не с нуля, а на основе уже имеющихся классов, используя их свойства и методы. При этом существенно снижается объем кодирования (не нужно два или более раз программировать одно и то же), уменьшается вероятность ошибок в программе.

Следуя [идеологии ООП](#), язык ТБ.Скрипт поддерживает наследование между двумя классами, один из которых выступает в качестве базового (родительского), а другой - в качестве наследника. Для описания родственных (наследственных) отношений классов в заголовках классов используется специальное ключевое слово, которое было для простоты опущено в синтаксисе заголовка, приведенном в предыдущем параграфе.

При описании производного класса применяется следующая форма записи:

```
Класс Унаследованный <ИмяБазовогоКласса> "<ПолноеНазвание>"
    [, редактор <ИмяКлассаРедактируемойЗаписи>];
Class Inherited <BaseClassName> "<Description>"
    [, editor <EditedRecordClassName>];
```

Здесь

ИмяБазовогоКласса - имя базового класса (то есть название COD-файла с исходным кодом этого класса),
Унаследованный (Inherited) - ключевое слово, объявляющее описываемый класс наследником базового.

В классе-наследнике можно использовать все свойства и методы, описанные в базовом классе (подробнее о свойствах и методах см. разделы Свойства класса и объектов, Описание переменных, Процедуры и функции), а также во всех остальных классах-предках. Кроме того в производном классе допускается доопределять новые свойства и методы. Особенно интересно то, что переменные базового класса могут фактически содержать объекты производных классов.

Например, если есть базовый класс **ФизЛицо** и несколько производных от него классов - **Сотрудник**, **Покупатель**, **Дилер** - то возможно использовать одну переменную типа **ФизЛицо** для хранения и обработки экземпляров классов **Сотрудник**, **Покупатель** и **Дилер**. При этом, в случае необходимости, используется так называемый динамический вызов свойств. Рассмотрим, что это такое.

Из описания базового класса компилятор ТБ.Скрипт знает о его свойствах и может проверить правильность обращения к этим свойствам еще на стадии компиляции. Однако когда переменная типа базового класса содержит объект производного класса, она может использоваться для обращения к свойствам производного класса, которых нет в базовом. В этом случае компилятор уже не может проверить правильность программы при компиляции (так как заранее не известно, какого класса объект будет содержаться в переменной) и генерирует код, который обеспечивает так называемое позднее связывание - связывание объекта и его свойства на стадии исполнения программы. Это и есть динамический вызов свойств (динамическая диспетчеризация). Он имеет как положительные, так и отрицательные стороны. Среди основных преимуществ - возможность писать более изящные (менее громоздкие и более понятные) программы. К недостаткам следует отнести некоторое увеличение времени вызова динамических свойств и возможность допустить ошибку, которая не будет выявлена на стадии компиляции.

В производном классе разрешается определять свойства, имеющие идентичное описание со свойствами базового класса. При этом говорят, что свойства производного класса **перекрывают наследуемые**. Такие свойства называются виртуальными, а их наличие позволяет реализовать еще одну важную особенность ООП - **полиморфизм**. Рассмотрим суть полиморфизма на небольшом примере.

Пусть имеется базовый класс **ФизЛицо** (с описанием в файле ФизЛицо.COD), в котором определен метод **Обработать** с одним аргументом строкового типа : FUNC Обработать (Назначение:Строка): Логическое;

Создадим производный от класса **ФизЛицо** класс **Сотрудник**, записав в файле Сотрудник.COD следующий заголовок:

```
Класс Унаследованный ФизЛицо "Сотрудник нашей фирмы";
```

Если в классе **Сотрудник** также определить метод **Обработать** с одним строковым аргументом, то он перекроет метод базового класса. Аналогично можно описать производные классы **Покупатель** и **Дилер**, причем в каждом из них действия, производимые функцией **Обработать**, отличаются от других.

Предположим теперь, что где-то еще в проекте записан код программы:

```
Var X: ФизЛицо[]; -- физ.лица
Var C: Сотрудник;
Var П: Покупатель;
```

```

Var Д:Дилер;
-- создаем по одному экземпляру каждого производного класса
С = Сотрудник.Создать;
П = Покупатель.Создать;
Д = Дилер.Создать;
X[1] = С;
X[2] = П;
X[3] = Д;
for i=1..3 do
    X[i].Обработать ("Гороскоп");
end;

```

Здесь в цикле для каждого объекта массива **X** вызывается метод **Обработать**. Однако, как ни странно, для каждого типа объекта будет вызвана своя функция **Обработать**.

Еще раз подчеркнем, что для того чтобы метод был перекрыт, его двойник в производном классе должен иметь такое же имя и точно такой же список аргументов (их число, последовательность и типы). Правда, у этого правила есть небольшие расширяющие исключения.

Если тип возвращаемого значения или аргумента метода является объектным (то есть не относится к числу элементарных типов: целый, числовой, логический, строковый, дата), то в наследнике одноименный метод может возвращать значение или иметь аргументы, тип которых унаследован от типа функции или соответствующего аргумента функции базового класса.

Пример:

```

-- содержимое файла "База.COD"
Класс "База";
    Проц ВиртПроц (Параметр1 :ОбъектШаблона);
        . . .
    Конеч;
    Функ ВиртФунк (ДокКласс :Класс Запись) :Запись;
        . . .
    Конеч;
Конеч

Класс Унаследованный База "Наследник";
    Проц ВиртПроц (Параметр1 :Кнопка);
        . . .
    Конеч;
    Функ ВиртФунк (ДокКласс :Класс Документы.Накладные) :Документы.Накладные;
        . . .
    Конеч;
Конеч

```

Здесь объектный тип **Кнопка** является производным от класса **ОбъектШаблона**, а класс **Документы.Накладные** - производный от класса **Запись**.

Если из метода производного класса необходимо вызвать перекрытый метод базового класса, то используется запись вида:

```

Унаследованный <Вызов метода>;
Inherited <Method invocation>;

```

Например, продолжая вышерассмотренный случай классов **ФизЛицо** и **Сотрудник**, для вызова исходного метода **Обработать**, унаследованного от класса **ФизЛицо**, из описания класса **Сотрудник** необходимо написать:

```

Класс Унаследованный ФизЛицо "Сотрудник нашей фирмы";
func Обработать(Назначение:Строка): Логическое;
    return Унаследованный Обработать(Назначение);
end;

```

Перекрывать можно не только методы (процедуры и функции), но и поля (переменные) класса.

Классы, описанные в любом проекте Студии, являются производными от встроенного класса **ПользовательскийОбъект**. Напоминаем, что бланки и картотеки, описанные соответственно в разделах [Подсистема бланков](#) и [Подсистема картотек](#), также являются классами Студии, производными от класса [ПользовательскийОбъект](#), однако обладают некими специфическими внутренними качествами (реализуемыми внутри системы), что делает невозможным наследование пользовательского класса от пользовательского

бланка (или картотеки) и наоборот - наследование пользовательского бланка (или картотеки) от пользовательского класса. Иными словами, пользовательский ("чистый") класс, описание которого содержится в отдельном COD-файле, может быть унаследован только от "чистого" класса. То же самое верно и в отношении способности класса выступать в качестве базового.

Иерархия классов с наглядным представлением наследования, а также свойств и методов классов (о которых пойдет речь в следующем параграфе), отображается в специальном окне Студии, которое так и называется ["Иерархия классов"](#). Новые классы и последующие изменения их свойств отображаются в данном окне только после компиляции содержащих их модулей.

Описание класса

Описание класса начинается с заголовка, в котором в кавычках задается полное название (комментарий) класса, представляющее собой произвольную текстовую строку. Напомним, что в отличие от названия, идентификатор класса соответствует правилам составления идентификаторов. Иногда после названия через запятую указываются некоторые необязательные параметры, такие как модификатор бланка-редактора.

После заголовка описываются глобальные переменные класса, а также процедуры и функции, задающие алгоритмы изменения и взаимосвязей используемых переменных. Описания переменных и процедур могут перемежаться.

Ключевые слова при описании класса записываются русскими или латинскими буквами - везде в данном руководстве приводятся оба варианта. Список всех ключевых слов ТБ.Скрипт приведен в Приложении. Любое ключевое слово может быть записано с произвольной капитализацией, то есть символами верхнего и нижнего регистра - компилятор ТБ.Скрипт не учитывает регистр. Однако хороший стиль программирования требует, чтобы все ключевые слова записывались единообразно в рамках одного проекта: например, только строчными буквами или с заглавной.

Завершается описание класса ключевым словом КОНЕЦ (без точки с запятой).

В контексте Студии существует две разновидности классов: встроенные и пользовательские. Встроенные классы реализованы внутри самой Студии и не могут быть изменены прикладным разработчиком. Пользовательские классы реализуются, в основном, на языке ТБ.Скрипт в виде логически взаимосвязанных алгоритмов определенного назначения, определяющих свойства и методы работы описываемых сущностей. Некоторые классы Студии, например, классы документов, формируются на основе MTL-описания. Они рассматриваются в разделе, посвященном картотекам. Здесь же речь пойдет о программировании классов, описываемых на языке ТБ.Скрипт.

Классы включают в себя процедуры, функции, переменные и константы. Все вместе эти свойства формируют программные интерфейсы для взаимодействия классов и построения единой программы.

В зависимости от сущности класса, он может состоять из нескольких представлений (или срезов), взаимодействующих и дополняющих друг друга. Например, бланк состоит из исходного кода, хранящегося в модуле (COD-файл) и определяющего логику его работы, и шаблона (TPL-файл), задающего внешний вид и структуру бланка. В свою очередь, картотека кроме модуля и шаблона имеет дополнительно BRO-файл с характерными только для картотек настройками.

Один из вариантов описания класса - так сказать "чистый" класс - имеет только одно - алгоритмическое - представление в виде исходного кода. Такой класс целиком содержится в отдельном COD-файле с текстом на ТБ.Скрипт. Как правило, в виде таких автономных классов реализуются обобщенные или вспомогательные алгоритмы, к которым обращаются из других классов, в том числе бланков или картотек. В терминах Студии класс, реализованный в виде одного COD-файла, называется библиотечным (или пользовательским) классом.

Далее в этой главе будут рассмотрены общие принципы программирования пользовательских классов, что, фактически, эквивалентно написанию COD-файлов. Следовательно, рассмотренные здесь приемы носят фундаментальный характер, поскольку COD-файлы используются при программировании большинства других классов Студии - бланков и картотек.

Пример использования синонимов

```
-- файл A.COD
class "Первый" synonym First;
InClass public
X Synonym Икс : Numeric = 10.0;
proc Test1;
  Trace(ClassName + 'Hello!');
  Second.Test1; -- эквивалентно вызову B.Test1
end;
-- Test2 принимает объект класса B
proc Test2 (Obj : Second);
  Obj.Тест1; -- эквивалентно Obj.Test1
end;
end

-- файл B.COD
class "Второй" synonym Second;
InClass public
proc Test1 synonym Тест1;
  Trace(ClassName + 'Hello!');
end;
proc Test2 synonym Тест2;
  -- выводим имя класса A
  Trace(First.ClassName);
  -- выводим значение переменной A.X
  Trace(First.Икс);
end;
end
```

Регистрация классов в проекте

Все классы, используемые в работе на том или ином участке автоматизированного учета, должны быть зарегистрированы в проекте, соответствующем этому участку. Осуществляется это с помощью команд контекстного меню [редактора проекта](#).

Для добавления класса необходимо в иерархии объектов [окна редактора проекта](#) выделить ветвь "Библиотечные классы" и выполнить команду **Добавить (Ins)** контекстного меню. Когда в иерархии выделен определенный класс, в этом же меню доступны команды **Удалить (Del)** и **Переименовать (Enter)**, а также команда **Добавить папку (Alt+Ins)**. С помощью них можно удалить класс из проекта, изменить его название или создать папку для хранения группы классов.

В результате выполнения команды **Добавить** на экран выводится диалог "[Создание класса](#)", где пользователь должен ввести название класса.

После заголовка класса и вплоть до завершающего ключевого слова КОНЕЦ следует исходный текст класса. При этом часть кода может описывать свойства самого класса, а часть - конкретных объектов данного класса. Свойства класса можно использовать в проекте в любой момент, в то время как свойства объекта доступны только при указании экземпляра объекта. Это объясняется тем, что классы, будучи зарегистрированными в проекте, существуют постоянно, а объекты - могут (и должны) создаваться и уничтожаться в процессе выполнения алгоритма.

Для указания, к какому типу относится тот или иной фрагмент описания класса в языке ТБ.Скрипт применяются две директивы: **ВКлассе** (**InClass**) и **ВОбъекте** (**InObject**).

Директива **ВКлассе** задает начало фрагмента кода, описывающего свойства класса в целом. Такой фрагмент продолжается до конца файла или до тех пор, пока в тексте не встретится директива **ВОбъекте**. В свою очередь директива **ВОбъекте** задает начало фрагмента кода, описывающего свойства конкретного объекта класса. Такой фрагмент продолжается до конца файла или до тех пор, пока в тексте не встретится директива **ВКлассе**. По умолчанию, если после заголовка класса не стоит ни одна из директив, считается, что описываются свойства объекта (то есть подразумевается, что первая директива **ВОбъекте** опущена).

Директивы **ВКлассе** и **ВОбъекте** не могут встречаться в теле процедур или функций.

Каждый объект существует до тех пор, пока ссылка на него хранится в программе. Например, если указатель был присвоен локальной переменной, объект будет уничтожен при выходе из процедуры или функции. (Если есть необходимость удалить объект явным образом до выхода из текущего метода, то можно присвоить переменной, хранящей указатель на объект, значение **Nil** или **Пусто**.) Однако если объект был создан внутри функции и передан в качестве ее результата в вызывающую процедуру (функцию), где присвоен некоторой переменной, то уничтожение объекта откладывается. В этом случае время существования объекта определяется многими дополнительными факторами: например, если указатель был присвоен переменной, являющейся полем класса, то логика его работы зависит от того была ли принимающая переменная описана в разделе **InClass** или **InObject**, а кроме того - собственно от особенностей класса. В частности, свойства класса **Запись** сохраняются навечно (в базе данных), а свойства класса **Бланк** - только до конца текущей сессии.

Кроме того, в классах Студии введено понятие "область видимости" свойства или метода класса. Существуют два типа областей видимости, определяемых ключевыми словами **Лично** (**Private**) и **Публично** (**Public**). Данные ключевые слова образуют секции в описании класса, аналогично словам **InClass** и **InObject**.

Свойства, описанные в разделе **Public**, доступны для обращений как в том же классе, в котором они описаны, так и в других классах. Свойства, описанные в разделе **Private**, могут использоваться только в том же классе, в котором они описаны. По умолчанию все свойства класса считаются **public**.

Благодаря возможности варьировать область видимости каждого свойства или метода можно скрывать подробности реализации того или иного класса. Это позволяет на уровне компилятора запрещать обращение извне к методам и переменным класса, для этого не предназначенным, что помогает избежать ошибок, сделанных случайно или по незнанию, а также сколь угодно сильно изменять реализацию класса (private - методы), не касаясь его внешнего интерфейса (public - методов) и, следовательно, минимизировать изменения в классах, которые используют данный класс.

Кроме того, ограничение видимости позволяет оптимизировать процесс компиляции классов, зависящих друг от друга. Если раньше (в предыдущих версиях Турбо Бухгалтера) при изменении текста класса перекомпилировались все классы, зависящие от данного, то сейчас перекомпилируется только те зависящие классы, которые используют изменившийся внешний интерфейс (то есть состав public-полей или методов, набор параметров в public-методах, тип возвращаемого значения у public-функций).

Это означает, что классы, зависящие от некоторого другого класса, не будут перекомпилироваться, если в том классе сделаны следующие виды изменений:

- вставлены пустые строки;
- изменено тело процедуры или функции, а ее параметры остались неизменными;
- изменены параметры или тип возвращаемого значения у private-метода;
- добавлены или удалены private-методы;
- добавлены, удалены или изменены private-свойства.

Подводя итог вышесказанному, можно сформулировать следующие рекомендации: делайте как можно больше переменных и методов класса с областью видимости **private** и как можно меньше - с **public**. Это позволит четче контролировать обращения к свойствам класса, а также уменьшит число перекомпиляций и упростит согласование программных интерфейсов.

При программировании классов ТБ.Скрипт бывает необходимо получить ссылку на сам объект или класс. Для этих целей используется специальное ключевое слово **Self**. В методах области **InClass** это слово возвращает

ссылку на текущий класс, а в методах области **InObject** - ссылку на текущий объект. С помощью **Self** можно, в частности, обратиться к переменной объекта, перекрытой локальной переменной, или же предоставить указатель на текущий объект для тех функций, которые принимают параметр соответствующего объектного типа. Пример:

```
x : Timer; -- переменная типа Таймер
proc Button2Click (Sender :Button); -- по нажатию кнопки
  -- создаем таймер с текущим бланком в роли владельца
  x = Timer.Create(self);
end;
```

Написанная один раз программа исполняется много раз. Условия, в которых она работает, и наборы обрабатываемых ею данных могут различаться от сеанса к сеансу. Поэтому в процессе исполнения программы могут возникать ситуации, требующие особой обработки. Это, прежде всего, различного рода ошибки в данных или недоработки в реализуемых алгоритмах. Такие ситуации называются исключительными.

Часто понятие исключительной ситуации (исключения) ассоциируют с программной ошибкой. На самом деле ошибка - термин более узкий, носящий деструктивный характер. Исключительной является любая ситуация, выходящая за рамки логики алгоритма. В ряде случаев программа сама может "объявить" о возникновении исключительной ситуации, например, в том случае, если набор данных, предлагаемых для обработки, ошибочен. При этом сама программа работает правильно.

Язык ТБ.Скрипт позволяет обрабатывать исключительные ситуации в процедурах и функциях, а также предоставляет программисту возможность возбуждать собственные исключения. Делается это с помощью блоков защиты ресурсов и обработки ошибок:

[Блок защиты ресурсов](#)

[Блок обработки исключительных ситуаций](#)

[Вложенные блоки](#)

[Коды исключительных ситуаций](#)

[Оператор Проверка / Assert](#)

[Функция КодОшибки / ErrorCode](#)

[Функция ТекстОшибки / ErrorText](#)

[Оператор Возбудить / Raise](#)

Блок защиты ресурсов служит для выделения операторов, которые должны быть выполнены независимо от возникающих исключительных ситуаций.

Блок состоит из трех ключевых слов: ПОПЫТКА (англоязычный аналог TRY), ОКОНЧАНИЕ (FINALLY) и КОНЕЦ (END). Эти директивы разделяют все операторы, включенные в блок, на две части. Между словами ПОПЫТКА и ОКОНЧАНИЕ располагаются операторы - потенциальные источники исключительных ситуаций. Если в одном из них произойдет исключение, то исполнение обычной последовательности операторов прервется, однако те операторы, которые расположены во второй части блока - между ключевыми словам ОКОНЧАНИЕ и КОНЕЦ - будут исполнены.

Такие блоки используются для обеспечения исполнения тех операций, которые являются парными к уже исполненным операциям.

Приведем пример использования блока защиты ресурсов:

```
проц ОбработкаКартотеки;
перем Q :ЗАПРОС;
Q = ЗАПРОС.Создать([ "Пример.ДвижениеРесурсов" ]);
НачатьИзоляцию([ "Пример.ДвижениеРесурсов" ]);
попытка
  ПОКА НЕ Q.EOF ЦИКЛ
    -- ... обработка информации
    Q.Следующий;
  КОНЕЦ;
окончание
  ЗавершитьИзоляцию; -- выполнится всегда
конец;
конец;
```

В данном случае использование блока защиты ресурсов обеспечивает завершение изоляции независимо от того, сколько документов и насколько успешно обработано в цикле.

Например, если в процессе обработки информации произойдет программная ошибка или иная исключительная ситуация, то последовательное выполнение операторов прервется и сразу будет выполнена процедура **ЗавершитьИзоляцию**. Метод **Следующий** объекта **Q** в такой ситуации исполняться не будет.

Если исполнение процедуры не нарушалось никакими исключениями, то операторы, заключенные между ключевыми словами ОКОНЧАНИЕ и КОНЕЦ, будут выполнены после выполнения всех операторов в первой части блока.

Если исключительная ситуация произошла ранее, чем исполнился первый оператор из первой части блока защиты ресурсов, то ни один оператор ни из одной части блока исполнен не будет.

Если в приведенном выше примере исключительная ситуация возникнет в процедуре **НачатьИзоляцию**, то не будет выполнена ни обработка информации, ни процедура **ЗавершитьИзоляцию**. В этом нет смысла, поскольку возникновение исключения на стадии инициализации изоляции означает, что она не была правильно начата, следовательно, ее не нужно закрывать, а обрабатывать неизолированные данные по тем или иным причинам нельзя.

Таким образом, блок защиты ресурсов позволяет программисту обеспечивать корректное завершение работы процедуры в любых ситуациях, в том числе и исключительных.

Блок обработки исключительных ситуаций - это специальная синтаксическая конструкция языка ТБ.Скрипт, позволяющая адекватно изменить ход выполнения программы в случае ее случайного или управляемого (вынужденного) перехода в нештатное состояние. С данного блока программист имеет возможность определить тип и возможную причину возникновения исключительной ситуации и предусмотреть реакцию на нее.

По своей структуре данный блок подобен описанному в предыдущем разделе блоку защиты ресурсов и отличается от него только вторым ключевым словом: вместо ОКОНЧАНИЕ в нем используется ИСКЛЮЧЕНИЕ (EXCEPT).

Операторы, находящиеся во второй части данного блока, будут исполнены только в случае возникновения исключительной ситуации в процессе исполнения одного из операторов, включенных в первую часть блока. Если исключений не произошло, вторая часть данного блока не исполняется. В этом заключается отличие блока обработки исключительных ситуаций от блока защиты ресурсов.

Если исключительная ситуация произошла ранее, чем исполнился первый оператор из первой части блока защиты ресурсов, то ни один оператор ни из одной части блока исполнен не будет.

Пример использования блока обработки исключительных ситуаций:

```
проц ОбработкаКартотеки;
перем Q :ЗАПРОС;
  Q = ЗАПРОС.Создать ( [ "Пример.ДвижениеРесурсов" ] );
  НачатьТранзакцию ( [ "Пример.ДвижениеРесурсов" ] );
  попытка
    ПОКА НЕ Q.ЕОФ ЦИКЛ
      -- ... обработка информации и вычисления
      Q.Следующий;
    КОНЕЦ;
  ЗавершитьТранзакцию;
Исключение
  ОтменитьТранзакцию;
конец;
Q = nil; -- принудительное освобождение памяти, это делать не обязательно
конец;
```

В случае возникновения ошибки внутри вышеприведенного блока обработки исключительных ситуаций происходит вызов процедуры **ОтменитьТранзакцию**, что гарантирует возврат картотеки в то состояние, в котором она была до начала редактирования.

Если не предпринимать никаких специальных действий, то исключительная ситуация будет существовать до тех пор, пока программа не выполнит один из блоков обработки. По выходе из него программа восстанавливает обычную последовательность выполнения операторов.

Иными словами, если исключительная ситуация не обработана, то состояние ошибки сохраняется даже при возврате в процедуру или функцию, вызвавшую данную. Таким образом будет прерываться и "сворачиваться" работа всех процедур в цепочке вызовов, пока исключение не будет обработано или процесс исполнения не вернется к первой процедуре данной цепочки. Если исключение не будет обработано и в ней, то Студия выведет на экран окно с сообщением об ошибке.

В приведенном выше примере оператор `Q = nil` будет выполнен всегда, т.к. даже в случае возникновения исключения в любом месте кода с обработкой данных оно будет "погашено" после выполнения второй части блока с процедурой **ОтменитьТранзакцию**.

Иногда от программиста требуется не допустить восстановления нормального хода выполнения процедуры, например, в случае каких-то фатальных ошибок, делающих всю работу процедуры невозможной или бессмысленной. В таких случаях следует заканчивать вторую часть блока обработки исключительных ситуаций оператором ВОЗБУДИТЬ (англоязычный синоним - RAISE), который приводит к восстановлению исключительной ситуации. Данный оператор, по сути, имитирует внутри себя возникновение той же самой ошибки, которая обрабатывалась данным блоком.

Допускается использование блока любого вида внутри другого блока. Число вложенных друг в друга блоков может быть произвольным. При этом они должны последовательно заканчиваться, т.е. первое ключевое слово КОНЕЦ относится к внутреннему блоку и так далее.

Вложенные блоки позволяют реализовать тот алгоритм защиты ресурсов и обработки исключений, который требуется для данной конкретной процедуры и функции бланка.

Пример использования вложенных блоков:

```
ПРОЦ p1;
  BeginTransaction([Накладная, Счет]);
  TRY
    BeginIsolation([Счет]);
    TRY
      -- анализируем счета
      -- ...
    FINALLY
      EndIsolation;
    END;
    -- здесь изменяем накладные и счета
    -- ...
  EndTransaction; -- записали изменения в базу
EXCEPT
  AbortTransaction;
  Raise;
END;
КОНЕЦ;
```

В данном случае при возникновении исключительной ситуации при открытии транзакции, выполнение процедуры прекратится; если исключение возникнет в процессе открытия изоляции или изменения накладных и счетов, то уже открытая к этому моменту транзакция, будет отменена; наконец, если ошибка возникает в процессе анализа счетов, то будет не только отменена транзакция, но и корректно завершена изоляция.

Допустимо использовать вложенные блоки обоих типов в произвольных сочетаниях, например, два вложенных блока защиты ресурсов, блок обработки исключительных ситуаций внутри блока защиты ресурсов или наоборот.

В языке ТБ.Скрипт имеются две встроенные функции ErrorCode и ErrorText, не принадлежащие ни к одному из классов объектной модели. Они предназначены для получения информации о текущей ошибке. Функция ErrorCode имеет следующий прототип:

```
func КодОшибки: Целое;  
func ErrorCode :Integer;
```

Прототип функция ErrorText:

```
func ТекстОшибки: Строка;  
func ErrorText :String;
```

Данные функции могут использоваться только в разделе ИСКЛЮЧЕНИЕ..КОНЕЦ блока обработки исключительных ситуаций. Важно, что функция КодОшибки может выдавать 0 для ошибок, не имеющих номера или контекста помощи.

Ошибки могут быть сгенерированы программным способом с помощью процедуры [УстОшибку / SetError](#) класса Система.

Для эффективной обработки исключительных ситуаций необходимо иметь возможность различать виды исключений с тем, чтобы для каждого из них предусмотреть соответствующую обработку. Внутри блоков обработки исключительных ситуаций это возможно за счет использования стандартной функции [Код ошибки](#). Данная функция возвращает код текущей исключительной ситуации, т.е. номер, присвоенный ей разработчиками Студия.

Функцию **КодОшибки** допустимо использовать только внутри второй части блока обработки исключительных ситуаций между директивами ИСКЛЮЧЕНИЕ и КОНЕЦ.

Приведем пример использования кода для обработки исключительной ситуации. Данный пример иллюстрирует также применение ключевого слова ВОЗБУДИТЬ и использование вложенных блоков обработки исключительных ситуаций:

```
Проц Обработка;
  Q = ЗАПРОС.Создать(["Пример.ДвижениеРесурсов"]);
  BeginTransaction([Накладная, Счет]);
  Попытка
    Попытка
      ОбработкаИнформации(Q);
    Исключение
      ЕСЛИ (КодОшибки<>21857) then
        -- код 21857 - нельзя изменять информацию в картотеке
        -- данная ошибка не приводит к прерыванию процедуры
        -- во всех других исключительных ситуаций ошибку не гасим:
        ВОЗБУДИТЬ;
      ИЛСЕ; -- конец условного оператора
    Конеч;
  СформироватьОтчет;
  EndTransaction;
Исключение
  AbortTransaction;
Конеч;
Конеч;
```

В данном примере все исключительные ситуации, которые могут возникнуть при исполнении процедуры **ОбработкаИнформации**, разделяются на фатальные, приводящие к прерыванию работы, и не являющиеся таковыми. К последним отнесено исключение с кодом 21857 (запрет изменений в картотеке), к фатальным - все остальные. Соответствующим образом построена и обработка исключительной ситуации: если код ошибки отличается от 21857, то исключение восстанавливается ключевым словом ВОЗБУДИТЬ, в противном случае не делается никаких действий, и ошибка "гасится".

Следует обратить внимание на поведение процедуры **СформироватьОтчет**. Если исключение будет погашено, то данная процедура выполнится, в противном случае сразу после ключевого слова ВОЗБУДИТЬ будет исполняться **AbortTransaction**. Таким образом, процедура **СформироватьОтчет** будет выполнена только в случае безошибочной работы программы или возникновения исключения с кодом 21857.

Существует возможность создания новых кодов исключений для обозначения тех ситуаций, которые могут возникнуть при исполнении конкретной процедуры или функции. Такие исключения называются **пользовательскими**, и их коды должны быть целыми отрицательными числами.

Стандартную или пользовательскую исключительную ситуацию можно возбудить намеренно в любом месте программы с помощью специальной процедуры **УстОшибку**, принимающей два входных параметра: код ошибки и строковое пояснение (последнее из них необязательно). Далее обработка этого исключения проводится по тем же правилам, что и для ситуаций, сгенерированных ненамеренно. Если ни один из блоков обработки исключений не погасил данную ситуацию, на экран будет выдано окно, содержащее строку (или код ошибки), заданную в процедуре **УстОшибку**.

Пример использования пользовательского исключения:

```
проц Вычисления;
  попытка
    ЕСЛИ НЕ ВычисленияДопустимы :
      УстОшибку(-1, "Нельзя вычислять!");
    КОНЕЦ;
  Вычисления1;
  Вычисления2;
```

```

Вычисления3;
Исключение
    ЕСЛИ (КодОшибки = -1):
        ВОЗБУДИТЬ;
| ИСТИНА:
    Сообщение("Вычисления прерваны из-за возникшего исключения")
    КОНЕЦ;
конец;
ДополнительноеВычисление;
конец;

```

В этом примере логическая функция **ВычисленияДопустимы** определяет, можно ли применить реализованный в данной процедуре алгоритм к текущему набору данных. Если результатом функции является ЛОЖЬ, то предлагаемые данные невозможно обработать, т.е. налицо исключительная ситуация. Для такой ситуации заведен пользовательский код, равный -1, и она считается фатальной для данной процедуры, т.е. дополнительное вычисление осуществлено не будет.

Данное исключение может быть обработано, например, в процедуре или функции, из которой вызвана процедура **Вычисление**, или появится на экране в виде окна с текстом "Нельзя вычислять!".

Выделение в процессе обработки информации исключительных ситуаций и их нумерация с помощью пользовательских кодов позволяет реализовывать гибкие алгоритмы, устойчиво работающие в любых ситуациях, а также помогает избегать многих неприятных ошибок.

Если бы в предыдущем примере не была диагностирована исключительная ситуация, то могли бы возникнуть ошибки в самом процессе обработки данных, что привело бы к их повреждению или потере.

Иногда от программиста требуется не допустить восстановления нормального хода выполнения процедуры, например, в случае каких-то фатальных ошибок, делающих всю работу процедуры невозможной или бессмысленной. В таких случаях следует заканчивать вторую часть блока обработки исключительных ситуаций оператором ВОЗБУДИТЬ (англоязычный синоним RAISE), который приводит к восстановлению исключительной ситуации. Данный оператор, по сути, имитирует внутри себя возникновение той же самой ошибки, которая обрабатывалась данным блоком.

Ключевое слово ВОЗБУДИТЬ допустимо использовать только внутри второй части блока обработки исключительных ситуаций, между директивами ИСКЛЮЧЕНИЕ и КОНЕЦ.

Если исключительная ситуация возбуждена заново, состояние ошибки сохраняется даже при возврате в процедуру или функцию, вызвавшую данную. Таким образом будет прерываться и "сворачиваться" работа всех процедур в цепочке вызовов, пока исключение не будет обработано или процесс исполнения не вернется к первой процедуре данной цепочки. Если исключение не будет обработано и в ней, то Студия выведет на экран окно с сообщением об ошибке.

Использование ключевого слова ВОЗБУДИТЬ, как правило, имеет смысл, если в процедуре применяются вложенные блоки защиты ресурсов и обработки исключений.

Кроме того программист имеет возможность возбудить "мягкое" программное исключение с помощью процедуры Откат / Abort класса **Система**. В отличие от оператора ВОЗБУДИТЬ, откат можно производить из любого места кода, а не только из блока обработки исключительных ситуаций.

С целью обеспечения дополнительной проверки на корректность написания кода программы в языке ТБ.Скрипт имеется специальный оператор **Проверка|Assert**. С его помощью осуществляется контроль за выполнением критических операций, которые могут вызвать ошибку уже на стадии исполнения программы. Как правило, оператор **Assert** используется для отладки программы на стадии разработки.

Оператор имеет следующий синтаксис:

```
Проверка (Условие:Логическое [; Сообщение:Строка]);  
Assert(Condition:Logical [; Message:String]);
```

Если логическое выражение, заданное параметром **Условие**, принимает значение Ложь, то оператор выдает сообщение об ошибке и прерывает выполнение программы, генерируя исключительную ситуацию. По умолчанию, если второй (необязательный) параметр опущен, сообщение включает в себя имя файла с исходным кодом, в котором произошла ошибка, и номер строки в нем (причем даже если в [настройках проекта](#) задана компиляция без отладочной информации). Если второй параметр все-таки используется, то указанная в нем строка дополняет сообщение об ошибке. Например:

```
Функ Дебет (Выражение : Число) : Число;  
    Проверка (Выражение > 0, "Дебетовый остаток < 0");  
    ^Выр;  
Конец;
```

Обработку оператора **Проверка** можно отключить. Для этой цели в [настройках проекта](#) имеется флаг **Отладочная версия (самопроверки включены)**. Когда он включен, оператор **Проверка** работает, как описано выше. Если же флаг снят, то оператор игнорируется и никаких проверок не выполняется. Это дает возможность широко использовать самопроверки на стадии проектирования, а перед выпуском окончательной версии продукта, не меняя кода, просто отключить их.

Переменная - это область памяти компьютера, в которой хранится то или иное значение, та или иная информация. Однако принципиально важным является то, что данная область имеет собственное имя, по которому ее можно отличить от других и потому обращаться к ней для выполнения специфических действий над хранящимся там значением. Иными словами, переменная - это абстрактное понятие, обозначающее информационную единицу независимо от ее значения.

Подробное описание переменных приводится в следующих темах:

[Имя переменной](#)
[Тип переменной](#)
[Переменная-массив](#)
[Формулы в описании переменных](#)
[Значение по умолчанию для переменных](#)
[Константы](#)

Например, значением переменной "ФамилияСотрудника" может быть любая, заранее неизвестная фамилия. Однако, если в алгоритме сделана ссылка на эту переменную, то в дальнейшей его работе будет использовано текущее, конкретное значение переменной. Именно таким образом и обеспечивается универсальность алгоритма, дающая возможность эффективно обрабатывать много однотипных значений, в разные моменты времени хранящихся в одной и той же переменной.

При разработке описания класса следует иметь в виду, что переменные, используемые в алгоритмах, могут быть как глобальными, так и локальными. *Глобальная переменная* описывается в COD-файле непосредственно внутри блока КЛАСС..КОНЕЦ, то есть вне процедур и функций. Такая переменная доступна из всех процедур и функций класса. *Локальная переменная* описывается внутри процедуры или функции и доступна только внутри этой процедуры или функции.

Важно отметить, что переменные, описанные внутри фрагмента [ВКлассе](#), существуют всегда (если иное не указано с помощью специального модификатора, *см. ниже*): присвоенные им значения хранятся на диске и восстанавливаются каждый раз при запуске проекта. Такие переменные называются *хранимыми*. Переменные, описанные внутри фрагмента [ВОбъекте](#), существуют только в контексте объекта и для обращения к ним необходимо иметь ссылку на проинициализированный объект.

Общая схема описания переменных приведена ниже:

```
[Stored|Хранимая] [Var|Перем] <Идентификатор>  
[ <ПризнакМассива> ]  
[ ( Синоним|Synonym)  
  <ДопИдентификатор_1> [{,<ДопИдентификатор_i>}]  
] : <Тип> [ <ПризнакМассива> ]  
[[:= <Выражение>|<Константа>];
```

Описание глобальной переменной, как правило, начинается с имени переменной (идентификатора), после которого через двоеточие должно следовать описание ее типа. Затем после знака равенства может идти формула для вычисления значения переменной. Существует также возможность задать для переменной значение по умолчанию (если не задана формула) - оно записывается тоже после типа переменной, но должно быть отделено от него двумя символами ":=". Заканчивается описание переменной точкой с запятой.

Если перед именем глобальной переменной поставить ключевое слово **Перем (Var)**, то такая переменная перестает быть хранимой. Если перед словом **Перем (Var)** поставить еще один модификатор **Хранимая (Stored)**, то переменная вновь становится хранимой. Таким образом, описание глобальной переменной, предваренное парой ключевых слов **Хранимая Перем**, равноценно описанию без единого ключевого слова перед ее идентификатором. Для случая особого типа класса - [бланка](#), который имеет визуальную форму - имеет смысл отметить, что его закрытие сопровождается генерацией события OnClose, которое вызывается до записи хранимых переменных в файл.

Описание локальной переменной всегда начинается с ключевого слова **Перем (Var)**, после которого следуют идентификатор переменной, двоеточие и тип переменной. Указание формул и значений по умолчанию для локальных переменных недопустимо. Также нельзя предварять описание локальной переменной модификатором **Хранимая**.

После идентификатора переменной или после идентификатора типа можно указывать признак массива - пару квадратных скобок с заключенным в них при необходимости целым числом, задающим размерность массива. Отсутствие числа между скобками определяет одномерный массив (вектор). Если у переменной нет признака массива, она является скалярной, то есть содержит лишь одно значение.

Будучи описанной, переменная может использоваться в составе выражений, где упоминание ее идентификатора приводит к извлечению текущего значения переменной.

Значение по умолчанию для переменных

Разновидностью формулы для вычисления переменной является значение по умолчанию. Если для некоторой переменной задано значение по умолчанию, то оно присваивается переменной сразу после ее создания в памяти компьютера. Для переменных типа [InClass](#) это происходит в момент запуска проекта (когда регистрируется класс), а переменные [InObject](#) инициализируются значениями по умолчанию непосредственно после создания родительского объекта.

Впоследствии присвоенное по умолчанию значение можно изменять.

Отличием значения по умолчанию от формулы является меньшая жесткость связи. Если для переменной задана формула, то ее значение будет вычисляться постоянно при изменении значения формулы. Это может происходить в результате изменения входящих в формулу переменных или функций. Значение же по умолчанию присваивается лишь однажды.

Значение по умолчанию представляет собой выражение того же типа, что и переменная, и задается после знаков двоеточия и равенства:

```
u :Число := 12.95;
```

Для переменных типа "Дата" значения по умолчанию задаются в формате ДД.ММ.ГГ или ДД.ММ.ГГГГ, а для логических переменных - в формате ИСТИНА или ЛОЖЬ (допустимо использовать англоязычные эквиваленты TRUE и FALSE):

```
ДатаНачала :Дата := 01.01.99;
```

```
НачислятьНДС :Логическое := ИСТИНА;
```

Имя переменной - это строка, удовлетворяющая правилам записи [идентификаторов](#).

В качестве имен переменных нельзя использовать идентификаторы, которые совпадают с ключевыми словами. Список ключевых слов приведен в [Приложении](#).

Выбор имени для переменной - очень важный момент в программировании. Несмотря на то, что более или менее сложные бланки могут содержать десятки переменных, каждая из них должна иметь собственное, логически обоснованное имя.

Не следует использовать для переменных ничего не значащие имена, например, называть их последовательно буквами алфавита. Такая практика может привести к тому, что программа станет очень непонятной, поскольку нельзя будет без значительных усилий вспомнить, в какой переменной находится нужная информация. В то же время нужно избегать слишком длинных имен для переменных, поскольку они будут загромождать тексты программ.

Хорошим стилем в программировании считается использование в качестве переменных идентификаторов длиной не более 8-10 символов. Такие идентификаторы могут состоять из нескольких слов, возможно, сокращенных. В этом случае начало каждого слова следует начинать с заглавной буквы, а остальные буквы в идентификаторе сделать строчными.

Примеры имен переменных:

```
ДатаНачала;  
Оборот50;  
АдресКонтрагт;
```

Имя переменной задается в ее описании, которое в простейшем случае имеет синтаксис:

```
[Var|Перем] <Идентификатор> : <Тип>;  
Например:  
КнопкаОтмена : Кнопка;
```

Сведения о типах переменных приведены в теме ["Тип переменной"](#).

Следует отметить, что ключевое слово **Перем (Var)** по умолчанию предвзывает описание только тех переменных, которые определены [локально внутри процедур и функций](#). Так называемые глобальные переменные, существующие в контексте всего класса, могут описываться без этого ключевого слова, а если оно будет использовано, то такая глобальная переменная перестает быть хранимой, то есть ее значение не сохраняется между сессиями. Однако если перед **Перем (Var)**, открывающим описание глобальной переменной, вставить также и ключевое слово **Хранимая (Stored)**, то переменная вновь станет хранимой. Пример:

```
Stored Var Number : Integer;
```

```
Stored Var Number : Integer;
```

Кроме основного имени любая переменная может иметь произвольное количество синонимов, которые задаются после ее идентификатора с помощью ключевого слова **Синоним (Synonym)**:

```
[Var|Перем] <Идентификатор>  
[ ( Синоним|Synonym)  
  <ДопИдентификатор_1> [ { , <ДопИдентификатор_i> } ]  
] : <Тип>;  
Например:  
КнопкаОтмена Синоним ButtonCancel : Кнопка;  
Сумма Синоним Sum, Summa, Итого : Число;
```

Синонимы можно использовать для работы с переменной точно также, как ее основное имя, что иллюстрируется [примером](#).

Константы

Константа в противоположность переменной содержит лишь одно, раз и навсегда заданное значение. По аналогии с переменными константы имеют имя, позволяющее адресовываться к ним из программ на языке ТБ.Скрипт. Синтаксис описания константы следующий: вначале следует имя константы, затем, после двоеточия, идет тип и, наконец, после знака равенства задается значение соответствующего типа, которое и будет содержать константа. Как всегда, завершает описание точка с запятой. Пример:

```
СтандартнаяСтавкаНДС1 :Число = 1.99;
```

Константы можно использовать только в правых частях выражений. Попытка повторно присвоить ей значение вызывает ошибку на стадии компиляции программы.

Некоторые переменные представляют собой не единственное (так называемое скалярное) значение, а набор значений, называемый массивом. Массив состоит из элементов, каждый из которых эквивалентен отдельной переменной, однако для доступа к элементу указывается не только имя (одно и то же для всех элементов), но и индекс. Индекс определяет порядковый номер элемента в массиве. В языке ТБ.Скрипт элементы массивов нумеруются, начиная с 1.

Примером массива значений может служить переменная для хранения наименований товаров в бланке счета-фактуры. В этом случае заранее, в момент разработки бланка, неизвестно количество необходимых позиций в счете и, соответственно, сколько переменных нужно для хранения их названий. В этом случае правильнее завести одну переменную-массив, в которой будут храниться все наименования товаров, сколько бы их не входило в конкретный счет.

Описание переменной-массива отличается от обычного описания переменных наличием после идентификатора квадратных скобок. Например, описание

```
Товар[]: Строка;
```

задает переменную-массив с именем **Товар**, содержащую в себе произвольное количество пронумерованных значений. Полностью эквивалентно следующее описание:

```
Товар: Строка[];
```

То есть, признак массива - квадратные скобки - могут также стоять и после имени типа.

Количество элементов в массиве при описании массива не задается, оно автоматически изменяется по мере необходимости, то есть в процессе выполнения алгоритма.

Обращение к элементу массива производится по имени переменной и индексу - номеру нужного элемента, проставленному в квадратных скобках. Например, чтобы обратиться к четвертому значению массива **Товар**, следует написать так:

```
Товар[4]
```

Вышерассмотренный массив является одномерным, так как в нем используется лишь один индекс - номер названия товара. Однако массивы могут быть и многомерными. Например, легко представить массив, в котором хранятся цены для различных товаров, причем каждый товар имеет заранее неизвестное количество типов цен (розничная, мелкооптовая, оптовая, крупнооптовая и т.д.). Значит, массив должен быть двумерным: по одной размерности отсчитываются сами товары, а по другой - типы цен.

Для описания многомерных массивов используется следующий синтаксис.

```
ЦеныТоваров[<ЦелоеЧислоРазмерность>]: Число;
```

Здесь в квадратных скобках после имени массива указывается размерность массива. Пример описания двумерного массива:

```
ЦеныТоваров[2]: Число;
```

Для доступа к элементам многомерного массива в квадратных скобках после его имени задаются через запятую индексы по каждой из размерностей.

Например, для того чтобы получить крупнооптовую цену третьего товара требуется записать:

```
ЦеныТоваров[3, 4]
```

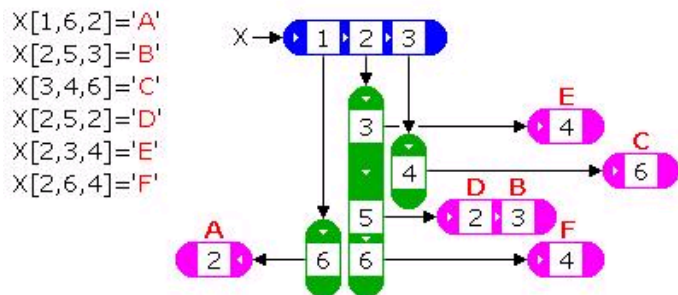
Здесь вначале указан индекс по товарам (3 - третий товар по списку), затем - по типам цен (4 - номер крупнооптовой цены). Важно, чтобы при заполнении и чтении массива использовался один и тот же порядок индексов. Дело в том, что способ индексации полностью определяется разработчиком и должен выбираться, исходя из удобства программирования алгоритма. Так, можно было бы тот же самый массив цен на товары организовать таким образом, чтобы тип цены шел по первому индексу, а наименование товара - по второму.

В языке ТБ.Скрипт массивы являются разреженными, то есть разработчик может, например, задать значения для второго и пятого элементов, а между ними фактически ничего не будет храниться в памяти, причем массив будет начинаться с индекса два. При попытке прочитать значение из непроинициализированного элемента система вернет значение **nil (пусто)**. Индексы могут быть и отрицательными.

Обычно говорят, что для двумерного массива одна размерность соответствует номеру строки (координата y), а другая - номеру столбца (координата x). Для трехмерного массива кроме строк и столбцов добавляет еще и своего рода "глубина" (z). И так далее. Однако на внутреннем уровне массивы ТБ.Скрипт хранятся не в виде гиперкуба, а иначе - списками. Для одномерного массива это привычный способ хранения - последовательность элементов (как говорилось выше, она может быть разреженной по индексу, поэтому в списке хранится не только значение элемента массива, но и его индекс).

Двумерный массив хранится в виде одномерного списка существующих индексов по первой размерности (т.е. это все индексы по первой размерности, по которым есть проинициализированные элементы), однако к каждому из элементов этого списка подключен список существующих индексов по второй размерности, причем у каждого элемента первого списка может быть свой собственный (отличный от других) список индексов по второй размерности. Здесь уже хранятся и значения элементов массива для соответствующей пары [первый индекс; второй индекс]. В случае трехмерного массива всё также начинается с одномерного списка, к элементам которого подключены одномерные списки по второй размерности, а к элементам этих списков - еще списки элементов, но уже по третьей размерности.

Данная концепция иллюстрируется следующим рисунком, где изображен трехмерный массив X:



Здесь синим обозначен список индексов по первой размерности, зелёным - связанные с ним списки по второй размерности, а малиновым - по третьей.

Для работы с массивами применяются [методы](#) из класса **Система**.

Синтаксис записи типов переменных совпадает с принятым в языке ТБ.Скрипт синтаксисом [типов данных](#).

Описание типа переменной начинается с двоеточия. Далее следует один из идентификаторов типов, который может быть либо одним из базовых типов программы, перечисленных ниже, либо названием класса, описанного в иерархии объектной модели Студии.

Тип	Русское обозначение	Английское обозначение
Числовой (с дробной частью)	Число Numeric,	Real
Целый	Целое, Целый	Integer
Строка	Строка String	
Логический	Логическое, Логический	Logical, Boolean
Дата	Дата	Date

Для типов переменных **Число**, **Целое**, **Строка**, **Логическое** и **Дата** *рекомендуется использовать полное обозначение типов*. Однобуквенные названия типов, определяемые первой буквой идентификатора типа, желательно не использовать.

Тип переменной определяет допустимое множество ее значений. Так, значениями *числового типа* являются действительные числа, т.е. числа, которые могут иметь дробную часть, отделяемую десятичной точкой. Над значениями числового типа определены арифметические операции, а также операции сравнения.

Значением *целого типа* является целое число. Переменная *логического типа* принимает значения Истина или Ложь.

Значения *строкового типа* представляют собой строки. Над строками можно производить операцию конкатенации ("склеивания" строк), а также операции сравнения.

Тип *"Дата"* описывает значение даты в формате ДД.ММ.ГГ или ДД.ММ.ГГГГ (с явным указанием столетия). Над датами допустимы операции сложения с целым числом (добавление дней), вычитания целого числа (вычитание дней), вычитания даты из даты (определение разницы в днях), а также операции сравнения.

Примеры описаний переменных:

```
ЧислоЕдиниц :Целое; Counter :Integer;          -- целые переменные
Сумма :Число; Money :Numeric;                  -- числовые переменные
Условие :Логическое; IfExist :Logical;         -- логические переменные
ДатаНачала :Дата; Period :Date;                -- переменные типа "Дата"
```

Типы данных **Целый**, **Число**, **Строка**, **Логическое** и **Дата** являются стандартными типами, т.е. используются во всех прикладных проектах - например, в описании типовых операций или структур данных (записей).

Остальные объектные типы Студии соответствуют классам, описанным в иерархии классов, которая включает как встроенные классы, реализованные внутри Студии, так и прикладные классы, реализованные на языке ТБ.Скрипт. Например, переменная типа **ОбъектШаблона** (или, иными словами, экземпляр объекта класса **ОбъектШаблона**) содержит в себе ссылку на компонент шаблона (визуальной формы бланка), такой как кнопка или рисунок. Обращаясь к переменной данного типа, можно манипулировать объектами, в том числе, изменять надпись на кнопках или выводить новую картинку на экран. Объектные типы и их использование подробно рассматриваются в *Справочнике по языку ТБ.Скрипт*.

Кроме типов, содержащих ссылку на объект некоторого класса, ТБ.Скрипт позволяет использовать типы со ссылками непосредственно на сам класс. Это необходимо в тех случаях, когда алгоритм пишется в достаточно обобщенном виде и предназначен для обработки данных из разных классов. Например, при создании запроса по информационной базе необходимо указать перечень классов записей, по которым будет строиться запрос. В данном случае используется массив классов записей. В описании переменных типа "ссылка на класс" перед именем класса пишут слово **Класс (Class)**. Так, переменная

```
х :Класс Запись;
```

может содержать ссылку на любой класс, производный от класса **Запись**.

Тип переменной **Запись** используется для хранения так называемых ссылок на записи (документы) в картотеках. Дело в том, что каждая запись картотеки имеет некоторый уникальный признак, отличающий ее от остальных и выделяющий в общем множестве записей. Этот признак можно сохранить в переменной типа **Запись**, чтобы, найдя нужную запись в картотеке один раз, впоследствии неоднократно к ней обращаться.

Поскольку значение ссылки является специфической информацией картотек, которая может устаревать

(например, запись, на которую указывала ссылка, была удалена), следует использовать тип **Запись** с осторожностью (например, выполнять дополнительные проверки).

Тип данных (или класс) : [Отчет](#) служит для организации доступа из программ, написанных на языке ТБ.Скрипт к информации, содержащейся во внутренних отчетах. Встроенный класс **Отчет** имеет богатый набор процедур и функций, позволяющих оперировать учетными данными и строить гибко настраиваемые отчеты, в том числе с расширением базового функционала системы и постобработкой результатов.

Пример описания переменных типов **Кнопка** и **Запись**:

```
КнопкаВызова :Кнопка;    -- переменная-объект шаблона
Поставщик    :Запись;    -- ссылка на запись в картотеке
```

В языке ТБ.Скрипт существует специальная константа **nil (пусто)**, которая сопоставима с любым типом и представляет собой нулевое (пустое) значение соответствующего типа.

С переменными могут быть связаны *формулы* для их вычисления. Наличие формул позволяет решить общие и частные задачи программирования на ТБ.Скрипт, например, упростить структуру программы за счет сокращения числа функций (так как некоторые функции можно заменить формулами) или облегчить процесс заполнения бланков. Если известно, что одна переменная однозначно зависит от другой, их можно связать с помощью формулы, и тогда при изменении ведущей переменной ведомая будет автоматически пересчитываться.

Например, если сумма НДС всегда составляет определенный процент от суммы платежа, есть смысл связать две содержащих их переменные формулой. Тогда достаточно будет ввести только сумму, а размер НДС вычислится автоматически.

Формула в описании класса - это выражение, в котором можно использовать стандартные функции Студии (см. [Справочник по объектному языку ТБ.Скрипт](#)), а также прикладные переменные и функции, описанные в исходных COD-файлах.

Формульное выражение, стоящее справа от знака "=", и переменная, которую обозначает идентификатор в левой части, должны иметь одинаковые типы данных.

Нельзя, например, строковой переменной присвоить числовое значение или целой переменной - значение типа "Дата".

Примеры переменных с заданными для них формулами:

```
X :Число = (A+B)/4;  
A :Целое = i-5;  
s :Строка = stroka+"строка";  
d :Дата = 19.06.97+5;  
l :Логическое = (Условие=0);
```

Следует заметить, что переменная **X** обязательно должна иметь числовой тип, т.к. в формуле присутствует операция деления и результат не обязательно окажется целым.

Стоит обратить особое внимание на последний пример. Переменная l имеет логический тип, и ее значением является результат проверки на равенство переменной **Условие** и нуля. Поскольку это равенство либо соблюдается, либо оказывается невыполненным, то результатом сравнения может быть либо ИСТИНА, либо ЛОЖЬ, т.е. значение логического типа. Результат логического типа дают все операции сравнения: "больше", "меньше", "не равно" и т.д.

Формула выражает связь между переменными (и полями в бланке) и, следовательно, вычисляется не один раз, а всегда, когда меняются значения переменных в правой части. Поэтому бессмысленно связывать с переменной две разных формулы. Кроме того, не имеют смысла следующие описания переменных:

```
B : Целое;  
C : Целое;  
A = B+C;  
C = A+B;
```

т.к. переменная C не может зависеть от значения самой себя.

Существует возможность в псевдовекторном виде записывать формулы для массивов (многозначных переменных или полей записи, связанной с бланком-редактором). Иными словами одна формула будет выполняться для всех элементов массива. Для этого необходимо использовать символ подстановки индекса \$ (или супериндекс). Например, формула

```
D :Integer[] = $;
```

присваивает всем элементам массива D их индексы, то есть массив содержит значения 1, 2, 3, 4 и т.д. Использовать такой массив в других участках исходного текста можно только поэлементно. В частности нельзя написать присваивание другому массиву:

```
Z = D;
```

Также нельзя передавать автовычисляемый массив в качестве параметров функций:

```
s = SumOfArray(D);
```

Однако можно написать:

```
Z[i] = D[i];
```

Основное назначение супериндекса - компактное представление выражений, в которых необходимо выполнить циклические вычисления по элементам массивов. Например, пусть в некотором классе документов под названием **Накладная** определена табличная часть (массив структур) **Товары**, содержащая поля с количеством и ценой товаров. Тогда в бланке-редакторе этого документа для вычисления сумм по каждой товарной позиции достаточно записать формулу:

```
Сумма :Numeric[] = Товары[$].Цена * Товары[$].Количество;
```

Разумеется, при изменении любого элемента выражения программа автоматически пересчитает результирующее значение, в данном случае - соответствующий элемент массива **Сумма**.

Еще раз напомним, что в формулах можно использовать методы встроенных классов, реализующих часто встречающиеся операции. Подробное описание этих классов приводится в [Справочнике по объектному языку ТБ.Скрипт](#). Кроме стандартных функций, реализованных внутри Студии, в классе доступны все функции, описанные пользователем как в нем, так и в других классах текущего проекта или его подпроектов.

Практически любая программа, ориентированная на решение более или менее сложной задачи методом настройки является установка тех или иных интерфейсных элементов программы в требуемое положение - например, переключение флагов в диалоговых окнах, выбор некоторых параметров путем выбора их из списков, ввод текста в поля ввода. Все подобные действия изменяют логику поведения программы, но лишь в строго ограниченных пределах, определяемых самой программой: ведь, фактически, от переключения флага сама программа не изменилась - просто в нее "зашиты" все возможные сценарии работы, выбор которых и осуществляется путем диалоговых настроек.

Студия, являясь средством разработки прикладных проектов довольно широкого профиля, поддерживает в дополнение к вышеупомянутому и другой, более мощный и гибкий способ настройки. Мы будем называть его языковым. Его суть в общих чертах сводится к следующему:

- пользователь пишет некоторый текст на специальном языке, понятном и ему, и компьютеру. Обычно такие языки состоят из ограниченного набора слов, называемых ключевыми *словами*. В Студии это слова русского языка или их англоязычные синонимы, поэтому получаемый текст более или менее похож на естественный язык, например, русский. Такой текст называют *программой*, а процесс его написания - *программированием*;
- Студия разбирает написанный текст программы и переводит его в некоторый внутренний вид, который затем используется компьютером. Например, на специальном языке, описанном далее в настоящем Руководстве, пользователь может создать для работы новый план счетов бухгалтерского учета. Студия, анализируя описание этого плана, отводит для каждого счета место в памяти компьютера, в которое будут заноситься остатки по данному счету, изменяющиеся в результате бухгалтерских операций. Процесс разбора текста программы и создания внутренних информационных структур называется *компиляцией*. Говорят, что Студия *компилирует* описание, созданное пользователем. Еще один термин - *компилятор* - обозначает часть Студии, предназначенную для разбора программных текстов. Процесс компиляции сопровождается выводом на экран характерного окна панели информации, на котором показывается имя разбираемого текстового файла и текущая обрабатываемая строка;
- по окончании процесса компиляции Студия способна следовать директивам, записанным в разобранных текстах. Важно понимать, что компьютер работает не напрямую с программами, написанными пользователем, а лишь с определенным "отражением", преобразованием этих программ, полученным в результате компиляции. Поэтому изменения в текстах воспринимаются Студией не непосредственно в момент их произведения, а только после *перекompиляции*. Вместе с тем Студия старается контролировать все пользовательские изменения и выполнять компиляцию только тех файлов, которые этого требуют.

Таким образом, языковой способ заключается в программировании, то есть написании управляющих текстов на языке, понятном Студии. Поскольку при написании программы в систему включается значительный элемент творчества, привносимый ее автором, то эффективность работы Студии под управлением таких специально написанных текстов резко повышается, и появляется возможность выполнять практически любые задачи, стоящие перед конкретным участком автоматизации.

Начать руководство по программированию, несомненно, следует со знакомства с терминологией, которая и вводится в следующих разделах.

[Терминология программирования](#)

[Исходный текст](#)

[Идентификатор](#)

[Переменная](#)

[Тип данных](#)

[Алгоритм](#)

[Оператор](#)

[Принцип нисходящего программирования](#)

[Понятие о программных ошибках](#)

[Советы начинающим](#)

Следующим важным понятием, без которого практически невозможно вести речь о программировании, является понятие алгоритма. Алгоритм - это записанная тем или иным образом последовательность инструкций, следуя которым, можно достичь заданного результата, например, вычислить сумму двух чисел или произвести расчет балансовых показателей. Алгоритм должен быть написан таким образом, чтобы отвечать следующим требованиям:

- быть понятным исполнителю. В данном случае исполнителем является программа Студия, и алгоритм должен быть записан на одном из ее языков. Языки манипулирования данными специально предназначены для записи алгоритмов;
- достигать своей цели за обозримое время. Если расчет заходит в тупик или начинает идти по кругу, говорят, что программа "зациклилась". В таком "зацикленном" состоянии программа и компьютер могут находиться неограниченно долгое время;
- быть применимым к определенному классу задач. Нет смысла делать сложный алгоритм под какой-то конкретный набор данных. Программирование - это сложная работа, и ее эффективность заключается именно в многократном применении единожды написанных алгоритмов. Поэтому следует писать программы так, чтобы они в состоянии были много раз решать однородные задачи. В то же время не следует стараться создать универсальный алгоритм, если это приводит к его сильному усложнению. Для особых случаев или для отдельных классов задач разумнее написать разные алгоритмы.

Языковые средства Студии - это, в основном, средства записи алгоритмов. Например, язык типовых операций позволяет записывать сколь угодно сложные алгоритмы формирования бухгалтерских проводок, а язык ТБ.Скрипт - алгоритмы работы с информацией, заключенных в первичных документах. Поэтому возможности Студии нужно рассматривать не сами по себе, а в разрезе их применения для записи алгоритмов, отражающих реальные правила выполнения тех или иных действий на автоматизируемом участке учета.

Одним из ключевых понятий любого языкового средства является понятие *идентификатора*. Идентификатор - это имя того или иного объекта, записанное по определенным правилам. Идентификатор состоит из букв английского и русского алфавита, цифр и знаков подчеркивания. Начинаться идентификатор должен обязательно с буквы или знака подчеркивания.

Примеры правильных идентификаторов:

```
Иванов
Кладовщик_Сидоркин
Счет60
_НачалоПрограммыНомер2
```

Следующие идентификаторы записаны с ошибками:

```
Сергей Сергеевич -- нельзя использовать пробелы
Dolce&Gabbana    -- нельзя использовать спецсимволы
60йСчет          -- нельзя начинать идентификатор с цифры
```

При анализе исходных текстов Студия не делает различий между заглавными и строчными буквами. Поэтому идентификаторы

```
ПРЕДМЕТ,
Предмет
и
ПредМет
```

считаются одинаковыми. Однако хороший стиль программирования требует, чтобы все идентификаторы в рамках одного проекта записывались единообразно.

Идентификатор может быть *простым* или *квалифицированным*, т.е. состоять из нескольких простых идентификаторов, разделенных точками. Именно поэтому квалифицированный идентификатор еще называют *составным*.

Как правило, такие идентификаторы обозначают объекты, являющиеся элементами каких-то общностей или групп. Например, идентификатор

Накладные.ТТН

может означать класс документов "Товарно-транспортная накладная", входящий в группу классов "Накладные". Зачастую в программистском обиходе длинное и неудобное при произношении слово "идентификатор" заменяется на синоним "имя", т.е. говорят "*имя объекта*". Эта замена является достаточно справедливой, и слово "имя" как синоним термина "идентификатор" используется в ряде случаев и в настоящем *Руководстве*.

Исходный текст

Термин *исходный текст* применяется для обозначения текстового файла, в котором записан текст программы. Такой текст называется исходным, поскольку он является основой для того или иного поведения Студии. В случае удаления исходного текста все созданные на его основе данные также будут потеряны или утратят смысл.

Например, исходный текст бланка первичного документа находится в текстовом файле, несмотря на то, что сам бланк в процессе работы с ним представляет собой некую электронную таблицу. Изменения в этом текстовом файле приводят к изменению поведения бланка, а удаление исходного текста делает бланк недоступным для пользователя.

Оператор

Алгоритмы, а точнее, их запись на встроенных языках Студии, должны из чего-то состоять, как, например, библиотека состоит из книг. Такой элементарный компонент алгоритма называется в программировании оператором. Само название оператора подчеркивает его активность и говорит о том, что он способен каким-то образом воздействовать на данные, которые обрабатывает алгоритм, или на сам ход обработки. Действительно, часть операторов обрабатывает информацию, например, складывает несколько чисел, а некоторые операторы управляют поведением алгоритмов, например, определяют, сколько раз и при каких условиях следует выполнить то или иное действие. Более подробно об операторах рассказывается в соответствующем [разделе](#) главы об общих принципах программирования на ТБ.Скрипт. Сейчас же важно понять, что алгоритм состоит из последовательности действий, каждое из которых представлено каким-либо оператором.

Оператор, как правило, состоит из одного или нескольких ключевых слов и содержит так называемые выражения.

Ключевое слово - это слово, использование которого в исходном тексте на языке программирования зарезервировано для правильного с логической точки зрения форматирования программы. То есть, ключевые слова можно использовать только по их прямому назначению, определенному семантикой языка программирования. Любое другое применение ключевого слова (например, в качестве идентификатора) запрещено. Перечень ключевых слов ТБ.Скрипт приведен в [Приложении](#).

[Выражение](#) - это отдельная языковая конструкция (часть программы), записанная в соответствии с синтаксисом языка программирования. Именно выражение дополняет оператор инструкциями по обработке конкретных данных (например, переменных, о которых мы уже знаем).

Идентификаторы используются для задания имен различным объектам Студии. Самый, наверное, распространенный вид такого объекта - это так называемая переменная. Переменная - это ячейка памяти, в которой хранится то или иное значение. Каждая переменная имеет свой идентификатор, по которому программа может к ней обращаться, т.е. брать текущее хранящееся в переменной значение и использовать его в своих расчетах. Иными словами, идентификатор переменной, входящий в какое-либо [выражение](#), заменяется при выполнении программы текущим значением переменной, которое и участвует в расчете.

Следует четко различать саму переменную, т.е. ее идентификатор, и значение, в ней хранящееся. Идентификатор есть имя переменной, которое используется в программе; идентификатор переменной не изменяется. Значение же есть та информация, которая хранится в переменной в определенный момент времени. Значение переменной может при необходимости изменяться, поэтому она и называется переменной.

Например, в переменной с именем (идентификатором) **НазваниеФирмы** содержится юридическое название предприятия. Если вдруг фирма будет переименована, то достаточно будет только изменить значение данной переменной, и все документы будут формироваться уже с новым названием. Это возможно потому, что при формировании документов используется ссылка на идентификатор данной переменной, а он остался неизменным.

Собственно, именно использование переменных вместо конкретных значений и делает возможным само программирование - написание наборов инструкций, манипулирующих абстрактными данными и последующее их использование для конкретных расчетов.

Более подробно о переменных рассказывается в разделе [Описание переменных](#).

"Человеку свойственно ошибаться". Эта древняя мудрость отражает вечное, к сожалению, правило, согласно которому редкому человеку доступно выполнение какого-либо действия без ошибок. Особенно это касается столь сложного вида деятельности, как написание программ. Даже опытные программисты неизбежно допускают ошибки, вероятность которых тем больше, чем сложнее программа.

Вместе с тем даже самая малая ошибка способна нанести серьезный урон деятельности предприятия, тем более, если эта программа используется для автоматизации бухгалтерского учета. Неверные данные в отчетных документах, убытки от неправильного планирования хозяйственной деятельности, штрафные санкции - это лишь малая часть последствий, которые может повлечь за собой такая ошибка.

Поэтому ошибки в программах следует рассматривать не как какой-то редкий и случайный фактор, а как объективную закономерность. Следовательно, необходимо разобраться в природе программных ошибок, их классификации и методах устранения.

Ошибкой в программировании называют нарушение логики работы программы, связанное с неправильностями в ее составлении, влияющими на получаемые результаты, или с возникновением непредусмотренной ситуации. Существует широкий спектр причин, приводящих к ошибкам: от невнимательности в использовании в языковых средствах до неправильного подсчета ситуаций, которые могут возникнуть при работе программы. Однако все ошибки, независимо от порождающих их причин, делятся по времени возникновения на две большие категории: ошибки компиляции и ошибки времени исполнения.

Ошибки компиляции - это класс ошибок, которые наиболее легко обнаружить и исправить. Их отыскивает компилятор Студии, когда обрабатывает программные тексты. При обнаружении такой ошибки открывается окно встроенного текстового редактора с загруженным в него файлом с программой, курсор устанавливается на место ошибки, а в строке состояния появляется пояснение, описывающее суть возникшей ошибочной ситуации. До тех пор, пока все ошибки данного класса не исправлены, текст программы не будет скомпилирован и "принят к исполнению" Студией.

Как правило, ошибки компиляции - это формальные ошибки, заключающиеся в нарушении при составлении программы каких-либо лексических или синтаксических правил. У языка программирования, как и у любого другого, есть свой лексикон - набор допустимых слов, и свой синтаксис, то есть правила записи текстов, понимаемые Студией. Нарушение этих правил или использование слов, не входящих в лексикон данного языка, приводит к тому, что программный текст становится "нечитабельным", и компилятор не способен в нем разобраться. В этом случае и появляется ошибка компиляции.

Например, в тексте использовано слово ЗАКОНЧИТЬ вместо слова КОНЕЦ. Казалось бы, с точки зрения русского языка разница небольшая. Однако в лексиконе Студии нет слова ЗАКОНЧИТЬ, и она не знает, что оно означает. Встретившись с незнакомым словом, компилятор текста остановится и определит, что произошла ошибка "Неизвестное ключевое слово ЗАКОНЧИТЬ".

Даже если в программе использованы только допустимые слова, они могут быть записаны с нарушением синтаксических правил, то есть не так, как этого требует язык. В этом случае Студия не сможет "понять" смысла всего высказывания, несмотря на то, что каждое из слов по отдельности является верным.

Например, описание функции должно заканчиваться ключевым словом КОНЕЦ. Если это слово будет пропущено, то Студия не сможет определить, где заканчивается одна функция и начинается следующая, даже если в остальном текст написан правильно. Поэтому отсутствие ключевого слова КОНЕЦ приведет к возникновению ошибки.

Еще одной распространенной причиной, приводящей к ошибкам компиляции, являются нарушения в программной логике, которые делают текст программы бессмысленным. Как правило, появление таких ошибок есть результат невнимательности или недопонимания каких-то механизмов языка.

Например, в программе делается попытка записать слово "БАРАБАН" в переменную целого типа. Каким может быть результат такого действия? Не удивительно, что Студия в этой ситуации тоже попадает в тупик и определяет ее как ошибку "Несоответствие типов "ЦЕЛОЕ" и "СТРОКА".

Все вышеперечисленные, а также другие, более редкие варианты ошибок компиляции могут быть найдены и исправлены еще на этапе обработки текста программы Студией за счет встроенных в нее мощных средств анализа текстов.

Когда программный текст скомпилирован, Студия может использовать написанные в нем инструкции в своей работе. Однако может случиться, что записанная по всем правилам программа работает непредусмотренным образом. Компилятор не в состоянии отследить такие ошибки, поскольку с формальной точки зрения текст верен: в нем использованы только допустимые ключевые слова и нет бессмысленных конструкций. Выявить такие ошибки можно только в процессе работы программы, поэтому их называют ошибками времени исполнения.

Например, при описании типовой операции неправильно указана корреспонденция счетов. Если при этом

формальных нарушений правил записи допущено не было, то компилятор такое описание "пропустит", т.е. преобразует в свой внутренний вид и будет использовать. И только сформировав отчет по проводкам, можно обнаружить результат данной ошибки.

Ошибки времени исполнения - наиболее опасный класс дефектов работы программы. Эти ошибки могут проявляться как явно (на экран будет выведено соответствующее сообщение), так и в скрытом виде, не приводя к видимым результатам. Программа с такими ошибками может производить впечатление работающей правильно.

Существует грустная программистская шутка: "Если программа работает верно, значит, ее ошибки уравнивают одна другую".

Борьба с появлением ошибок - одна из самых важных задач программирования. Разработчики Студии постарались предоставить своим пользователям максимально удобный и дружелюбный инструмент для разработки надежных программных текстов, включающий в себя мощный текстовый редактор, интеллектуальный компилятор, отладчик, позволяющий наблюдать за работой программы "изнутри". Однако лучшим средством против ошибок является хорошее знание программистом языковых средств, имеющихся в его распоряжении, правильное и эффективное их использование.

Программы предназначены для того, чтобы облегчать человеку выполнение сложной, объемной и регулярно повторяющейся работы. Это означает, что сама по себе программа есть очень сложная и, возможно, громоздкая структура, состоящая из большого числа инструкций (директив) компьютеру, что и как ему делать.

Чтобы облегчить процесс создания сложных алгоритмов, в программировании выработано несколько принципов проектирования структуры программы. Одним из наиболее эффективных является так называемый принцип *"нисходящего программирования"*. Он состоит в разделении сложного алгоритма на части с постепенным увеличением степени подробности рассмотрения каждой из частей.

Разделение сложного алгоритма на отдельные части (в программировании они называются процедурами или функциями) - очень мощное средство, помогающее создавать большие и правильно работающие программы.

Проектирование такого алгоритма нужно начинать "сверху вниз": сначала определить, что он должен делать и в какой последовательности, не вдаваясь в подробности того, как он должен это делать. Каждый выделенный шаг алгоритма нужно назвать. Это название будет именем будущей процедуры, реализующей этот шаг. Таким образом, на самом верхнем уровне программа (алгоритм) представляет собой последовательность вызовов других процедур.

Когда разделение на процедуры проведено, можно приступить к конкретизации каждого шага алгоритма, "заполняя" его операторами. Не исключено, что некоторые из этих шагов также окажутся достаточно сложными и их в свою очередь будет целесообразно разделить на части.

Приведем пример простого алгоритма, созданного методом "нисходящего программирования". Пусть нужно задать программе правила рисования человечка из простейших геометрических фигур. Нужная нам фигурка состоит из головы, туловища, двух рук и двух ног. Соответственно верхний уровень алгоритма будет состоять из процедур "Нарисовать голову", "Нарисовать туловище", "Нарисовать руки" и "Нарисовать ноги". Цели каждой из этих процедур достаточно простые, поэтому можно сосредоточиться на их полной и правильной реализации. Например, процедура "Нарисовать голову" будет состоять из директивы "Нарисовать большую окружность", двух директив "Нарисовать маленькую окружность" для глаз, директивы "Нарисовать дугу", изображающую рот фигурки и т.д.

Важно, что на момент написания высокоуровневых процедур тех элементов, из которых они будут в конечном итоге состоять, еще нет, они только объявляются и перечисляются. Поэтому данный метод и называется "нисходящим программированием", когда на проблему смотрят сначала сверху, постепенно доходя до ее глубин и записывая короткие и несложные алгоритмы, которые объединяются для решения общей задачи.

Следование некоторым приведенным ниже рекомендациям позволит облегчить процесс написания правильных и надежных программных текстов.

- Перед написанием нового текста следует четко сформулировать для себя цель будущей программы.
- Написание программы оправдано только в том случае, если достижение поставленной цели невозможно с помощью неязыковых средств [Настройка Студии](#).
- При написании программы следует уделить особое внимание процессу проектирования, т.е. составления схемы будущей программы. Особенно это важно при создании достаточно сложной прикладной системы, затрагивающей несколько взаимодействующих средств Студии: структуры учета, типовых операций, бланков, картотек и т.д.
- Написание текста целесообразно начинать с самого простого: создания пустого бланка, описания простейшего плана счетов и т.д.
- Программам и их элементам необходимо давать осмысленные названия. Старайтесь, чтобы составляемый текст был максимально похож на естественный и мог быть легко понят. Вставляйте [комментарии](#) в текст программ, чтобы не забывать смысл сложных языковых конструкций.
- Чаще компилируйте написанный текст. Компиляцию можно использовать как самое простое средство поиска явных ошибок.
- Текст программы рекомендуется сохранять на диске как можно чаще. Также целесообразно перед внесением изменений в работоспособный вариант текста, например, при необходимости его модификации, сохранять предыдущий текст под другим именем, чтобы в случае неудачи к нему можно было бы вернуться.
- Для исправления сделанных ошибок при редактировании текстов используйте для отмены исправлений команду "Отменить", которой обладает встроенный текстовый редактор Студии. Обязательно следует выключить флаг "Сброс при сохранении" из [диалога](#) "Настройки редактора", чтобы информация о сделанных изменениях не уничтожалась после каждой записи файла на диск.

Надеемся, что следование этим простым правилам окажет действенную помощь при составлении программных текстов на языках Студии и позволит воспользоваться всем богатством ее возможностей.

Как и в любом виде деятельности, в программировании принята своя терминология. Слова, обозначающие те или иные программистские понятия, заимствованы по большей части из обиходного языка, однако иногда в своем новом качестве они получают значение, отличное от общепринятого. Такая терминологическая путаница становится одним из основных препятствий, встающих перед начинающими программистами. В данном разделе мы попытаемся привести основные термины, используемые при программировании в Студии, и проиллюстрировать их примерами.

Вводимые далее термины и понятия активно используются в программистской практике, для общения специалистов, а также в руководствах и справочниках. Поэтому владение данной терминологией является одним из залогов успеха в использовании инструментальных средств Студии:

[Исходный текст](#)

[Идентификатор](#)

[Переменная](#)

[Тип данных](#)

[Алгоритм](#)

[Оператор](#)

Тип данных

Основной характеристикой переменной является *тип* хранимого в ней значения. *Типом данных* называется область значений, которые может принимать данная переменная, и допустимые над ней операции. Различают следующие основные типы данных:

- *целое число*;
- *действительное число* (число, имеющее десятичную часть);
- *строка* - последовательность символов;
- *дата* - день, месяц и год григорианского календаря;
- *логический* - возможны два значения данного типа: ИСТИНА или ЛОЖЬ.

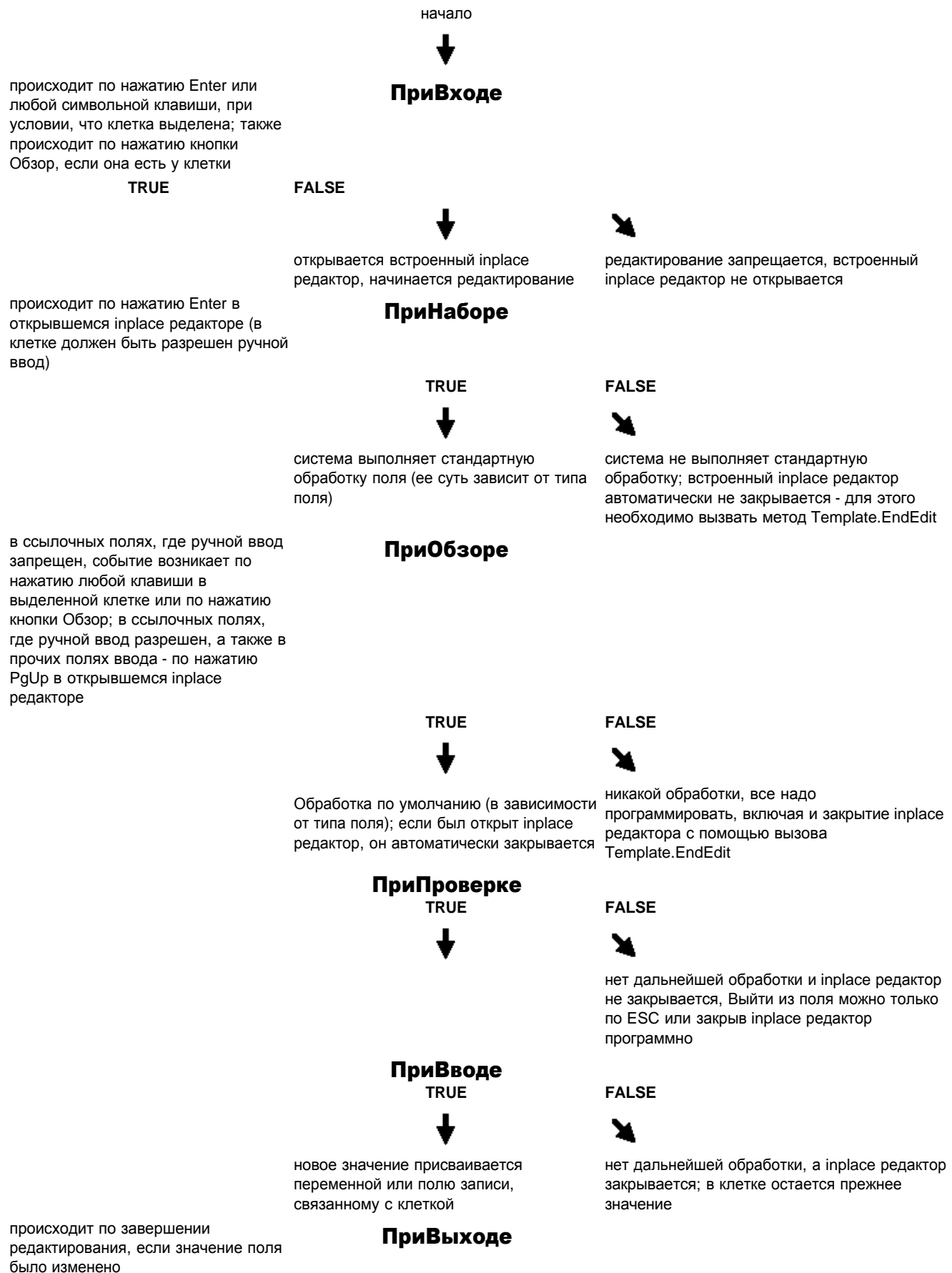
Например, число 21 имеет целый тип, а число 3.8 - действительный, поскольку имеет "восемь десятых". Значением строкового типа может быть, например, строка "Иванов Иван Иванович", типа дата - 21.09.91, а значением логического типа может служить результат сравнения пола сотрудника и мужского пола (в данном случае либо сравнение истинно, либо ложно).

Тип данных определяет поведение переменной и допустимость ее использования в выражениях. Например, из строки нельзя вычесть число, даты нельзя перемножать и т.д. Таким образом, тип определяет "родовые признаки" переменной и особенности ее поведения.

Более подробно о типах Студии изложено в теме [Стандартные типы данных](#).

В данном разделе с помощью схем описываются наиболее важные последовательности возникновения и обработки событий в объектах Студии. Такие классы, как **КлеткаШаблона**, **СтолбецКартотеки**, **Шаблон** имеют достаточно сложный механизм взаимодействия с прикладным кодом на уровне событий, что требует наглядного представления всей цепочки происходящих действий. Объекты большинства других классов имеют малое число событий (до 3), которые генерируются в виде одиночных событий, не объединенных в единую логическую цепочку. В связи с этим, представление таких событий в виде схем не требуется, а условия возникновения и назначение методов-обработчиков событий подробно изложены в соответствующих разделах Справочника по иерархии классов ТБ.Скрипт.

- [Последовательность событий в объектах КлеткаШаблона](#)
- [Последовательность событий в объектах Шаблон](#)
- [Последовательность событий в объектах СтолбецКартотеки](#)



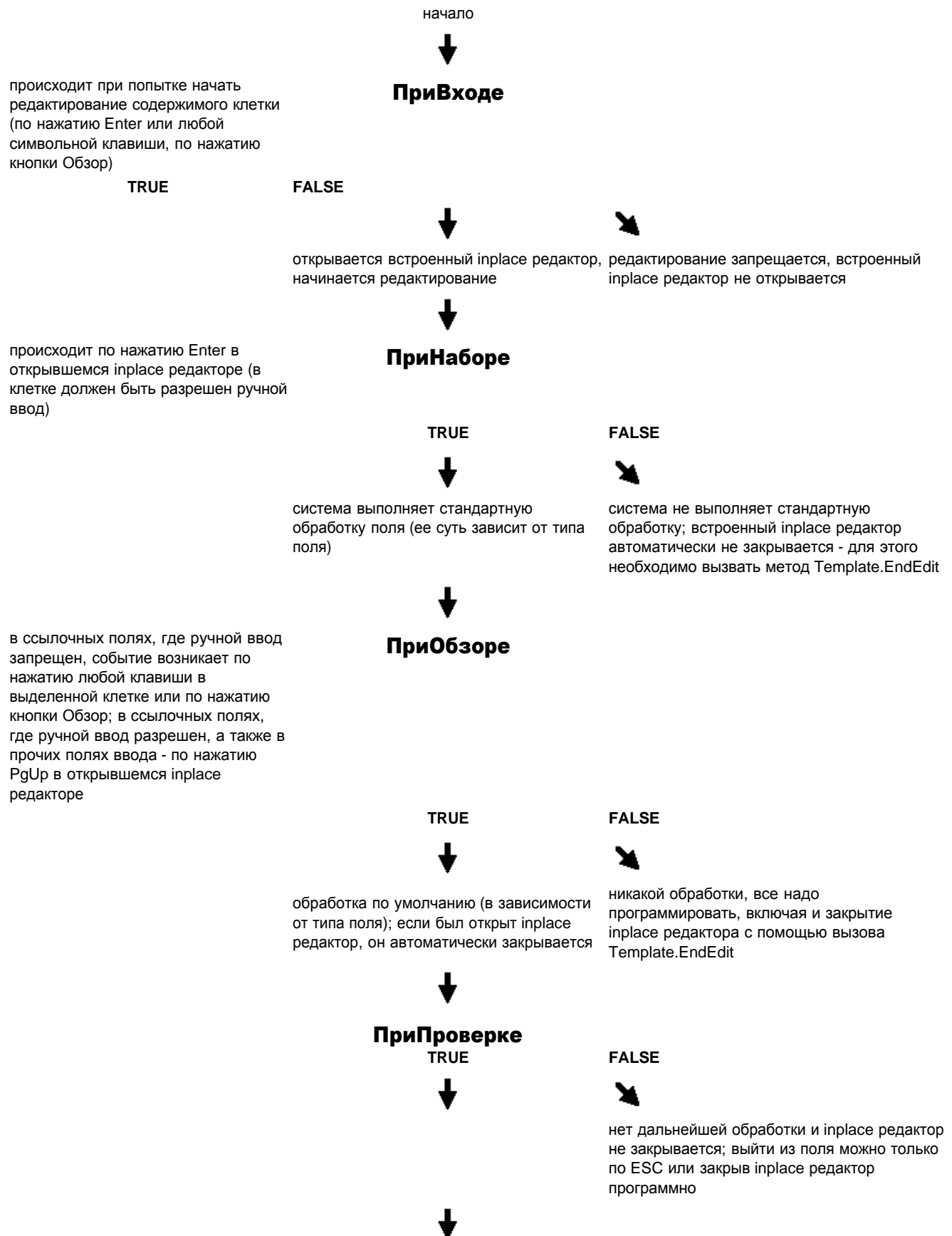


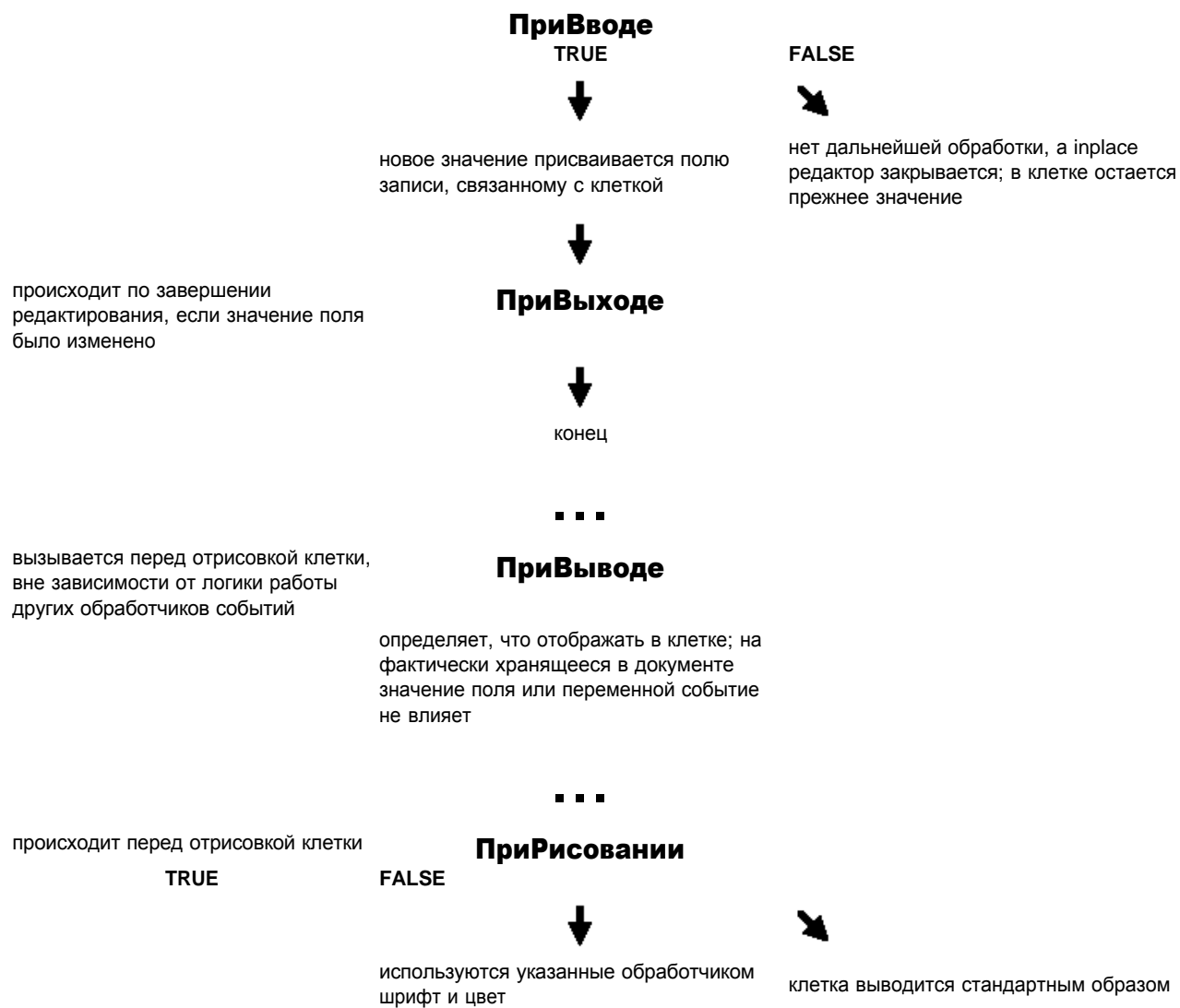
конец

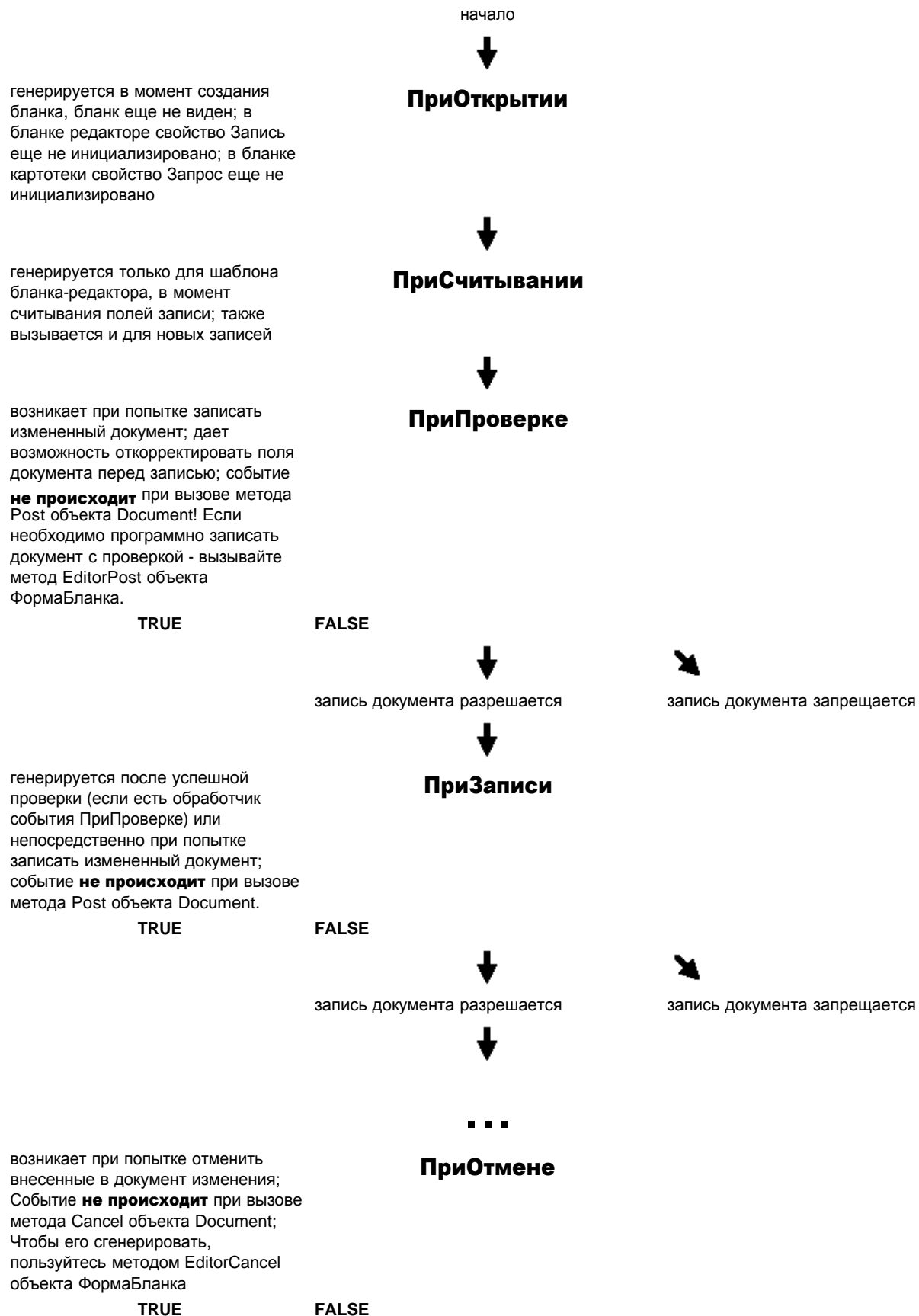
вызывается перед отрисовкой клетки,
вне зависимости от логики работы
других обработчиков событий

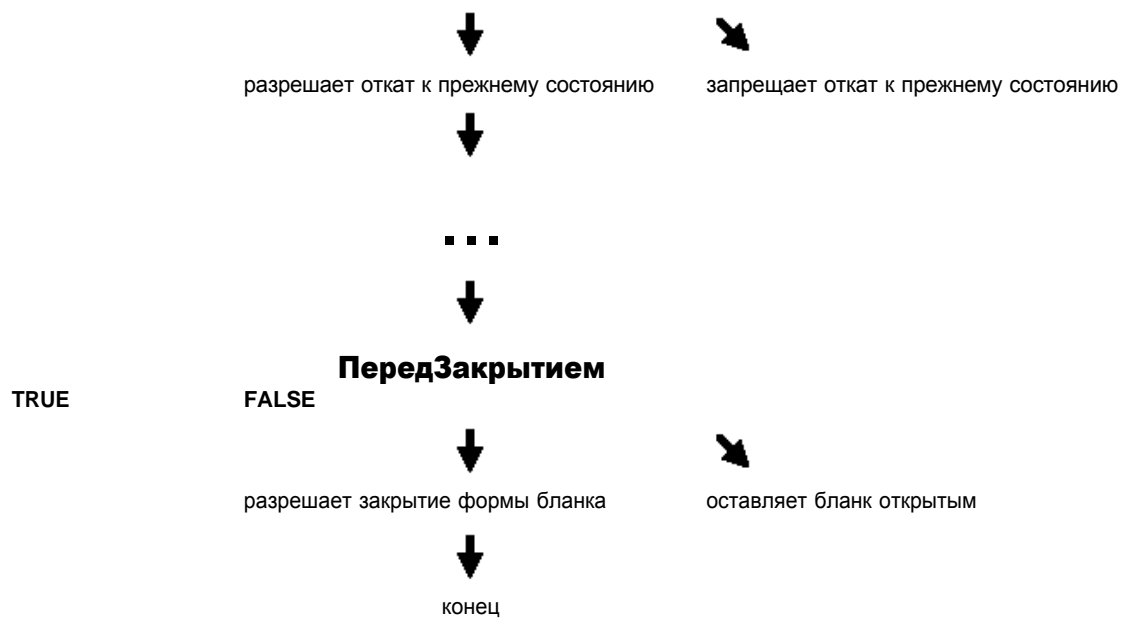
ПриВыводе

определяет, что отображать в клетке. На
фактически хранящееся в документе
значение поля или переменной событие
не влияет









Из обработчиков событий **ПриПроверке/ПриЗаписи/ПриОтмене** нельзя вызывать методы **EditorPost** и **EditorCancel**, так как это приведет к заикливанию. События **ПриПроверке** и **ПриЗаписи** имеют одинаковую семантику, поэтому достаточно обрабатывать лишь одно из них, хотя допустимо определить оба обработчика.

[Пример работы с DBF-файлами. Открытие и чтение DBF-файла](#)

[Пример работы с объектом Отчет](#)

[Пример расчета курсовых разниц](#)

[Пример работы с иерархическим отчетом](#)

[Пример работы с объектом Запрос](#)

[Пример просмотра документов из результатов запроса](#)

[Пример работы с секциями, столбцами, строками и клетками шаблонов](#)

[Пример обработчика события ПриНаборе в клетке шаблона](#)

[Пример поиска в картотеке](#)

[Пример использования объекта Импортер](#)

[Пример запроса полупроводок](#)

[Примеры запросов по аналитике](#)

[Пример запроса счетов](#)

[Пример разрешения конфликтов репликации](#)

Пример запроса полупроводок

```
class "TestHTrans";
inobject
  BegDate: Date;
  EndDate: Date;
  AccCond: String;
  ParamCond: String;

-- подготавливаем условия отбора и записываем их в переменные
...

-- запускаем запрос по полупроводкам
proc OnTestClicked(Sender :String);
  var q : HTransQuery;
  var h: HTrans;

  q = HTransQuery.Create(AccCond, ParamCond, BegDate, EndDate);
  while not q.Eof do
    h = q.Current;
    -- выводим информацию о каждой полупроводке
    Trace('Date'+Str(h.Date)+' '+if(h.IsDebet,'+', '-')+
      ' '+Str(h.Acc)+' Сум='+Str(h.Сум));
    q.Next;
  end;
end;

end;
```

Пример запроса счетов

```
var q: AccQuery;  
var a: Account;  
var n: integer;  
  
q = AccQuery.Create;  
q.AccPlan = 'БАЛАНС';  
q.AccMask = '5*';  
q.Select;  
  
-- навигационный подход  
n = 1;  
while not q.EOF do  
  a = q.Current;  
  Trace(Str(n)+' ' + Str(a));  
  q.Next;  
  n = n + 1;  
end;  
-- цикл с прямым доступом  
for n = 1..q.Count do  
  Trace(Str(n)+' ' + Str(q[n]));  
end;
```

Пример использования объекта Импортер

```
proc Import1BtnClick (Sender :Object);
var fName :String;
var ImportObj :Importer;
-- запрашиваем имя файла для импорта
if ChooseFile(fName, "Выбор файла", "Данные в формате tbc|*.tbc")
  <> cmOk then
  return;
end;

-- создаем объект
ImportObj = Importer.Create;

-- отключаем автоматическое соответствие
ImportObj.AutoCorr = False;
-- устанавливаем соответствие явным образом
ImportObj.AddCorrespondence('Test.DB1.Full', 'Full',
  ['Номер1 = Номер',
   'Дата = Дата',
   'Строка = Строка',
   'Число = Число',
   'Флаг = Флаг',
   'Докум = Докум',
   'Структ1.Номер1 = Структ1.Номер1',
   'Структ1.Дата1 = Структ1.Дата1',
   'Структ1.Строка1_1 = Структ1.Строка1',
   'Структ1.Докум1 = Структ1.Докум1']);

-- выполняем операцию импортирования
ImportObj.ImportFromFile(fName, '*.tbc');
end;
```

```
func ТМЦ_ПриНаборе(C:TemplateCell; Key:String; Value:String;
                  var NewValue:DocS.Справочники.Ресурс):Logical;
var Q :Query;
var Способ,РезультатВыбора:Integer;
var Найдено:Logical;
var OldValue,ГдеСтоим:DocS.Справочники.Ресурс;
var Crd:Справочники.Ресурс.ТМЦ;

if Key<>CHR(13) then -- ввод поисковой строки еще не завершен
  return Истина;
fi;
-- ввод поисковой строки завершен. Ищем.
OldValue = NewValue;
-- в зависимости от настроек программы ТМЦ будут
-- вводиться по коду или по названию
Способ = Def.НастройкаВводаСсылокНаТМЦ;
Q = Query.Create([DocS.Справочники.Ресурс]);
Q.РежимЗагрузкиПолей = СИС.Константы.mdNone;
-- для использования констант режимов загрузки полей
-- требуется, чтобы в списке подпроектов был указан проект СИС
Q.ЗагружаемыеПоля = if(Способ<3:"КОД","Наимен");
Q.Order = if(Способ<3:"КОД","Наимен");
Q.Select;
Найдено = Q.FindNearest(Value);
if (НЕ Найдено) И Способ=2 then
  ГдеСтоим=Q.Current;
  Q = Query.Create([DocS.Справочники.Ресурс]);
  Q.РежимЗагрузкиПолей = СИС.Константы.mdNone;
  Q.ЗагружаемыеПоля = "Наимен";
  Q.Order = "Наимен";
  Q.Select;
  Найдено = Q.FindNearest(Value);
elseif (НЕ Найдено) И Способ=4 then
  ГдеСтоим=Q.Current;
  Q = Query.Create([DocS.Справочники.Ресурс]);
  Q.РежимЗагрузкиПолей = СИС.Константы.mdNone;
  Q.ЗагружаемыеПоля = "КОД";
  Q.Order = "КОД";
  Q.Select;
  Найдено = Q.FindNearest(Value);
fi;
if Найдено then
  NewValue = Q.Current;
else
  Crd = Справочники.Ресурс.ТМЦ.Create;
  РезультатВыбора=Crd.ExecuteEx(ГдеСтоим,"");
  if РезультатВыбора=cmOK then
    NewValue=ГдеСтоим;
  fi;
fi;
Template.EndEdit(OldValue <> NewValue);
return Ложь;
end;
```

Пример поиска в картотеке

```
proc Search(SearchString: String);
var Found :logical;
var n :integer;
-- ищем подстроку по всей картотеке, с зависимостью от регистра
Found = Cardfile.Find(SearchString, 1, TRUE, FALSE, TRUE, TRUE);
-- подсчитываем все записи, удовлетворяющие условию
if Found then
  if ВопрДаОтказ ("Продолжить поиск?") = кмдОтказ then
    return;
  end;
  n = 1;
  while Cardfile.FindNext do
    n = n+1;
    if ВопрДаОтказ ("Продолжить поиск?") = кмдОтказ then
      break;
    end;
  end;
end;
message(n);
end;
```



```
Класс "БланкЮЛ", Редактор Пример.Контрагент;

-- кнопки для перехода к следующему/предыдущему документу
ButtonNext : Button;
ButtonPrev : Button;

proc ButtonNextPressed(O:String);
var Q:Query;
Q = Query.Create([Пример.Контрагент]);
Q.Select; -- строим запрос
-- делаем текущим документ, открытый в бланке-редакторе
Q.Current = Document;
-- переходим на следующий в запросе документ
Q.Next;
-- управляем доступностью кнопок в зависимости от
-- того, можно ли пролистывать документы дальше или нет
if Q.BOF then
    ButtonPrev.Enabled = FALSE;
else
    ButtonPrev.Enabled = TRUE;
end;
if Q.EOF then
    ButtonNext.Enabled = FALSE;
else
    ButtonNext.Enabled = TRUE;
end;
-- выводим текущий документ в бланк-редактор
Document = Q.Current;
end;

proc ButtonPrevPressed(O:String);
var Q:Query;
Q = Query.Create([Пример.Контрагент]);
Q.Select;
Q.Current = Document;
Q.Previous;
if Q.EOF then
    ButtonNext.Enabled = FALSE;
else
    ButtonNext.Enabled = TRUE;
end;
if Q.BOF then
    ButtonPrev.Enabled = FALSE;
else
    ButtonPrev.Enabled = TRUE;
end;
Document = Q.Current;
end;

end -- класса бланка-редактора
```

Пример работы с DBF-файлами. Открытие и чтение DBF-файла

-- чтение dbf-файла и сохранение информации в картотеке Price
proc ReadPrice(s:String);

```
var dbffil:DBFFile;  
var Zap:Tables.Price;  
var col:Integer;
```

```
-- открыть dbf с Dos-кодировкой  
dbffil=DBFFile.Create("goods.dbf","",0,true);  
col=0;  
try  
  while not dbffil.eof do -- пока dbf не кончился  
    Zap=Tables.Price.Create;  
  
    zap.Ean      = dbffil.Field["Ean"]; -- читать поля  
    zap.Plu      = dbffil.Field["Plu"];  
    zap.Name     = dbffil.Field["Name"];  
    zap.quantit= dbffil.Field["quantit"];  
    zap.PTax     = dbffil.Field["p_tax"];  
    zap.Cas      = dbffil.Field["Cas"];  
    zap.price    = dbffil.Field["price"];  
    col=col+1;  
    zap.Post;  
    dbffil.next; -- перейти к следующей записи  
  end;  
finally  
  Message('В картотеку "Price" загружено записей: '+Стр(col));  
end;  
end;
```

```
ОтчетТМЦ      :Report;
ГлубинаИерархии :Integer;

Шапка1        :Section;
Поз           :Section; -- секция с результатами
Итог1         :Section;

-- массивы, связанные с секцией Поз результатов отчета
Наименование  :String[];
Остаток       :Numeric[];
Стоимость     :Numeric[];
Уровень       :String[];

proc кнСформироватьПриНажатии(Sender :Button);
with ОтчетТМЦ do
  Build;
end;
ЗаполнитьМассив;
end;

proc ЗаполнитьМассив;
-- функция ЗаполнитьПодМассив заполняет все кадры
-- секции Поз результатами отчета
-- и возвращает количество кадров
Поз.Count = ЗаполнитьПодМассив(0,0);
end;

func ЗаполнитьПодМассив(Offset :Integer; Level :Integer): Integer;
var nTables :Integer;
var nRows   :Integer;
var nColumns:Integer;
var i,j,k,n :Integer;
var AddedCount:Integer;
var x :Integer;
var Symbol:String;

-- счетчик строк, добавленных в подтаблицах
AddedCount = 0;

with ОтчетТМЦ do
  nTables = TableCount;
  for i = 1.. nTables do
    CurTable = i; -- координата 1: таблица
    nRows = RowCount;
    for j = 1..nRows do
      CurRow = j; -- координата 2: строка
      -- далее берем из каждой колонки значения
      -- для заполнения массивов, отображаемых в секции Поз
      CurColumn = 1;
      Остаток[Offset+AddedCount+j] =
        UnitValue(ОтчетТМЦ.EndSaldo(Report.Roll, 1));
      CurColumn = ColumnCount;
      Стоимость[Offset+AddedCount+j] =
        UnitValue(ОтчетТМЦ.EndSaldo(Report.Roll, 2));
      if Остаток[Offset+AddedCount+j] <> 0 then
        Наименование[Offset+AddedCount+j] =
          ОтчетТМЦ.SplitValue[Report.rdRow].Описание;
      else
        Наименование[Offset+AddedCount+j] =
          ОтчетТМЦ.SplitValue[Report.rdRow];
      end;

      -- готовимся пометить группы специальными значками
      if CanOpen then -- это группа?
```

```

        Symbol = '1'; -- символ "открытая папка", если для колонки секции
                        -- выбран шрифт Wingdings
    else
        Symbol = '3'; -- символ "документ", если для колонки секции
                        -- выбран шрифт Wingdings
    end;
    -- делаем отступ по мере спуска вниз иерархии
    Уровень[Offset+AddedCount+j] = RepStr(' ',Level) + Symbol;

    if CanOpen then -- это группа?
        -- включен ли просмотр иерархии, т.е.
        -- нужно ли заходить в подгруппы?
        -- если да, то не достигли ли мы
        -- максимального заданного уровня?
        if (ГлубинаИерархии = 0) OR (Level < ГлубинаИерархии) then
            OpenGroup; -- входим в подгруппу
            x = ЗаполнитьПодМассив(Offset+AddedCount+j,Level+1);
            AddedCount = AddedCount + x;
            CloseGroup; -- возвращаемся назад
        end;
    end;
end; -- for j
end; -- for i
end;
return AddedCount+nRows;
end;

```

```
-- проверить наличие изделий, которые требуется поставить
-- по действующим счетам
func DoQuery(Filter: String): Numeric;
var Q : Query;
var Q2 : Query;
var Total : Numeric;
var product[] : Numeric;
var i : Integer;

Total = 0;

-- подготовить выборку по счетам
Q = Query.Create([Тест.Пример.Счет]);
Q.Order = "Дата";
Q.Filter = "ТипСчета=1"; -- 'Действующий счет'
Q.LoadingFieldsMode = СИС.Константы.mdNone;
-- для использования констант режимов загрузки полей
-- требуется, чтобы в списке подпроектов был указан проект СИС
Q.LoadingFields = "Дата;DocId;Контрагент;Товар";
Q.PacketSize = 50;
Q.Select; -- делаем выборку

-- подготовить выборку по изделиям
Q2 = Query.Create([Тест.Пример.Склад]);
Q2.Order = "Название";
Q2.Filter = Filter; -- задаем набор видов техники
Q2.LoadingFieldsMode = СИС.Константы.mdNone;
Q2.LoadingFields = "DocId;Название";
Q2.PacketSize = 100;
Q2.Select;

for i=1..Q2.Count do
    product[i] = Q2.Current.Количество;
    Q2.Next;
end;

while not Q.EOF do -- для каждого счета
    Q2.First;
    i = 1;
    while not Q2.EOF do -- для каждого изделия
        if Q.Current.Товар = Q2.Current.Название then
            Total = Total + Q.Current.Total;
            product[i] = product[i] - Q.Current.Total;
            if product[i] < 0 then
                Q2.Current.НужноЗакупить = ИСТИНА;
            end;
        end;
        Q2.Next;
        i = i + 1;
    end;
    Q.Next;
end;

return Total;
end;
```

```
-- импортируем класс Отчет из ядра Студии,  
-- чтобы при обращении к свойствам отчета  
-- можно было опускать название класса, например  
-- rdTab вместо Report.rdTab  
Import Kernel classes Report;  
  
proc Button2OnClick(Sender :Button);  
  var x:Report;  
  var nTables: integer;  
  var nRows: integer;  
  var nColumns: integer;  
  var i,j,k,n: integer;  
  
  x = Report.Create("Отчет1");  
  x.Build;  
  try  
    nTables = x.TableCount;  
  except  
    Message("Отчет не был построен. Исправьте настройки отчета.");  
    return;  
  end;  
  for i=1..nTables do  
    x.CurTable = i;  
    nRows = x.RowCount;  
    try  
      trace("Таблица:"+Str(x.SplitValue[rdTab]));  
    except  
      -- разбиения на таблицы нет  
    end;  
    for j=1..nRows do  
      x.CurRow = j;  
      nColumns = x.ColumnCount;  
      try  
        trace("  Строка:"+Str(x.SplitValue[rdRow]));  
      except  
        -- нет разбиения на строки  
        trace("  Строка");  
      end;  
      for k=1..nColumns do  
        x.CurColumn = k;  
        try  
          trace("    Столбец:"+Str(x.SplitValue[rdCol]));  
        except  
          -- разбиения на колонки нет  
          x.CurColumn = 0;  
        end;  
        for n=1..x.MeasureCount do  
          trace("      Оборот("+x.MeasureName[n]+")="+Str(x.Turn(Roll,n)));  
          trace("      НачальныйОстаток("+x.MeasureName[n]+")="+Str(x.BegSaldo(Roll,n)));  
          trace("      КонечныйОстаток("+x.MeasureName[n]+")="+Str(x.EndSaldo(Roll,n)));  
        end;  
      end;  
    end;  
    x.CurRow = 0; -- итоги по столбцам  
    trace("Итоги по столбцам");  
    for k=1..nColumns do  
      x.CurColumn = k;  
      try  
        trace("  Итого:"+Str(x.SplitValue[rdCol]));  
      except  
        -- нет колонок  
        x.CurColumn = 0;  
      end;  
    end;  
    for n=1..x.MeasureCount do
```

```
        trace("      Оборот("+x.MeasureName[n]+"")="+Str(x.Turn(Roll,n)));
        trace("      НачальныйОстаток("+x.MeasureName[n]+"")="+Str(x.BegSaldo(Roll,n)));
        trace("      КонечныйОстаток("+x.MeasureName[n]+"")="+Str(x.EndSaldo(Roll,n)));
    end;
end;
end;
end;
```

```
-- процедура импорта шаблона в лист MS Excel
proc MakeSheet(var Sheet:AutoObject);
var Sec:TemplateSection;
var Col:TemplateColumn;
var Row:TemplateRow;
var ThisCell:TemplateCell;
var ThisExcelCell:AutoObject;
var i,j,k:Integer;
var ExcelRow:Integer;
var LastPrintedRow,LastPrintedSec:Integer;
var LnSt:Integer;
var m:Integer; --!!!

ExcelRow=0; LastPrintedRow=0; LastPrintedSec=0;
-- начинаем цикл по секциям шаблона
for i=1..Template.SectionsCount do
-- для оптимизации быстродействия сохраняем текущую
-- секцию в переменной Sec
Sec=Template.Section[i];
-- обрабатываем секцию только в том случае,
-- если она должна выводиться на печать
if Sec.Printed then
-- цикл по всем клеткам секции
for j=1..Sec.RowsCount*Sec.Count do
-- вычисляем номер строки
m=mod(j,Sec.RowsCount);
m=if(m=0:Sec.RowsCount,m);
-- для оптимизации быстродействия
-- сохраняем ссылку на текущую строку в Row
Row=Sec.Row[m];
-- обрабатываем строку только в том случае,
-- если она должна выводиться на печать
if Row.Printed:
-- цикл по колонкам секции
for k=1..Sec.ColumnsCount do
Col=Sec.Column[k];
if Col.Printed then
if LastPrintedSec<>i ИЛИ LastPrintedRow<>j:
ExcelRow=ExcelRow+1;
LastPrintedSec=i; LastPrintedRow=j;
Sheet.Rows[ExcelRow].RowHeight=Row.Height*2.54;
fi;
-- находим текущую клетку шаблона
ThisCell=Sec.Cell[k,j];
if -- экспортируем клетку, только если...
-- в ней число или дата и она непуста...
((ThisCell.FieldType=3 ИЛИ ThisCell.FieldType=4)
И ThisCell.Value<>NIL) ИЛИ ThisCell.Text<>"
-- или если есть хотя бы одна рамка
ИЛИ ThisCell.Border[1]<>0 ИЛИ ThisCell.Border[2]<>0
ИЛИ ThisCell.Border[3]<>0 ИЛИ ThisCell.Border[4]<>0 then
-- находим текущую ячейку в Excel
ThisExcelCell=Sheet.Cells[ExcelRow,k];
-- записываем в ячейку Excel значение или текст
-- из клетки шаблона
if ThisCell.FieldType=3 ИЛИ ThisCell.FieldType=4 then
ThisExcelCell.Value=ThisCell.Value;
else
ThisExcelCell.Value=ThisCell.Text;
fi;
-- устанавливаем параметры шрифта ячеек в Excel
ThisExcelCell.HorizontalAlignment=ThisCell.AlignMent+1;
ThisExcelCell.Font.Name=ThisCell.Font.Name;
ThisExcelCell.Font.Size=ThisCell.Font.Size;
```



```

ThisExcelCell.Font.Bold=ThisCell.Font.Bold;
ThisExcelCell.Font.Italic=ThisCell.Font.Italic;
-- устанавливаем тип рамок для ячеек в Excel
LnSt=ThisCell.Border[1];
if LnSt>0 then
  ThisExcelCell.Borders[1].Linestyle=1;
  ThisExcelCell.Borders[1].Weight=LnSt+1;
fi;
LnSt=ThisCell.Border[2];
if LnSt>0 then
  ThisExcelCell.Borders[3].Linestyle=1;
  ThisExcelCell.Borders[3].Weight=LnSt+1;
fi;
LnSt=ThisCell.Border[3];
if LnSt>0 then
  ThisExcelCell.Borders[2].Linestyle=1;
  ThisExcelCell.Borders[2].Weight=LnSt+1;
fi;
LnSt=ThisCell.Border[4];
if LnSt>0 then
  ThisExcelCell.Borders[4].Linestyle=1;
  ThisExcelCell.Borders[4].Weight=LnSt+1;
fi;
fi;
fi;
end;
fi;
end;
fi;
end;
end;

```

```
proc Button2OnClick(Sender :Button);
var vQuery :ReplConflQuery;
var vDocsQuery :Query;
var vOpQuery :Query;
var vConflDoc :Record;
var pItem :Новый.Документы.Товар;
var pOpDoc :Новый.Документы.ПриходРасход;
var vDoc :Record;
var ret :Integer;
var i :Integer;

-- Создаем запрос на конфликтные записи
-- класса Новый.Документы.Товар
vQuery = ReplConflQuery.Create([Новый.Документы.Товар]);
vQuery.Select;

-- Если нет конфликтов репликации - сообщаем
if vQuery.Count = 0 then
  Message('Нет конфликтов репликации');
  return;
end;
trace(vQuery.Count);

-- Начинаем транзакцию по всем классам записей
BeginTransaction;
try
  while not vQuery.Eof do
    vConflDoc = vQuery.Current; -- документ из пришедшей реплики
    if vQuery.CurrentConflType = ReplConflQuery.Changed then
      -- Есть две версии одного и того же документа,
      -- получаем документ из текущей базы
      vDoc = vQuery.CurrentConflSource;
      -- запрашиваем пользователя
      ret = Enquiry ("Конфликт",
        "Запись модифицирована на сервере-отправителе и на сервере-получателе",
        ["Принять пришедшую запись", "Подправить пришедшую запись", "Подправить имеющуюся запись", "Отказаться от пришедшей записи"]);
      if ret = 1 then
        -- Принимаем пришедшую правку
        vQuery.AcceptCurrentConfl;
      elseif ret = 2 then
        -- Копируем содержимое пришедшей записи в базу
        vDoc.Edit;
        vDoc.Assign(vConflDoc);
        -- Изменяем значение некоторого поля
        -- (здесь - просто записываем в него случайное число)
        vDoc.Код = random(2147483647);
        vDoc.Post;
        -- Этот конфликт уже разрешен, отменяем его
        vQuery.AbortCurrentConfl;
      elseif ret = 3 then
        vDoc.Edit;
        -- Изменяем некоторые поля, взяв значения из пришедшей версии
        vDoc.Код = vConflDoc.Код;
        -- ...
        vDoc.Post;
        -- Теперь пришедшая версия записи не нужна,
        -- помечаем этот конфликт как разрешенный
        vQuery.AbortCurrentConfl;
      elseif ret = 4 then
        -- Просто отказываемся от принятия изменений
        vQuery.AbortCurrentConfl;
      end;
    elseif vQuery.CurrentConflType = ReplConflQuery.UniqueFieldsValues then
      -- Нарушено условие уникальности
      -- В записи о товаре уникально поле Наименование и это означает,
      -- что заведено два товара с одним именем.

      -- Проверим, нет ли в других документах (полученных также с новым
      -- пакетом репликации) ссылок на конфликтующий товар

      -- Необходимо:
      -- 1. открыть запрос с фильтром на уникальное поле,
      --    напр. 'Наименование='+vConflDoc.Наименование
      -- 2. найти нужную запись
      -- 3. перенаправить все ссылочные поля, которые
      --    ссылаются на запись vConflDoc, на найденную запись
      vDocsQuery = Query.Create([Новый.Документы.Товар]);
      vDocsQuery.Filter = 'Наименование="'+vConflDoc.Наименование+'";
      vDocsQuery.Select;
      while not vDocsQuery.Eof do
        pItem = Новый.Документы.Товар(vDocsQuery.Current);
        -- делаем запрос по операционным документам
        vOpQuery = Query.Create([Новый.Документы.ПриходРасход]);
        -- ищем документы, в табличной части "Позиции" которых
        -- имеются в поле "Товар" ссылки на некорректный товар
```

```

-- используем квантор Exists
vOpQuery.Filter = 'Позиции.Exists(Товар='+Str(vConflDoc)+'');
vOpQuery.Select;
while not vOpQuery.EOF do
  pOpDoc = Новый.Документы.ПриходРасход(vOpQuery.Current);
  -- цикл по всем строкам табличной части
  for i = 1..pOpDoc.Позиции.Количество do
    -- если это дефектная ссылка...
    if pOpDoc.Позиции[i].Товар = vConflDoc then
      -- меняем ее на имеющийся одноименный товар
      pOpDoc.Позиции[i].Товар = pItem;
    end;
  end;
  vOpQuery.Next;
end;
vDocsQuery.Next;
end;
-- запись о полученном в реплике конфликтующем товаре
-- следует удалить из списка конфликтов репликации:
-- это можно сделать различными способами, рассмотренными
-- в ветке конфликтов типа ReplConflQuery.Changed, но
-- в данном случае мы просто отказываемся от пришедшей
-- записи о товаре, так как в операционных документах уже
-- все ссылки перенаправлены на аналогичный товар, имеющийся
-- в базе
vQuery.AbortCurrentConfl;
end;
vQuery.Next;
end;
--Завершить транзакцию на конфликты репликации
EndTransaction;
except
  --Отменить транзакцию на конфликты репликации
  AbortTransaction;
  raise;
end;
end;
end;

```

-- Пример использования отчетов в типовых операциях
-- для оптимизации расчета курсовых разниц
-- Строится внутренняя структура данных (невидимый отчет), на
-- основании которой получаем информацию для расчета курсовых разниц

```
class "";  
  
import classes Common, спрВалюта;  
  
oper КурсоваяРазница = "Курсовые разницы" ;  
  
var нач: numeric;  
var кон: numeric;  
var локВалюта: спрВалюта;  
var локОст : numeric;  
VAR F :Report;  
VAR i :Integer;  
VAR j :Integer;  
VAR Инд44 :Integer;  
VAR кол_002 :Numeric;  
VAR ост_002 :Numeric;  
VAR Инд002 :Integer;  
  
var нач1: numeric;  
var кон1: numeric;  
var разница: numeric;  
  
F= Report.Create(Report.byTurn);  
with F do  
  ПланСчетов = 'Баланс';  
  ФильтрСчетов =  
'76.2_нач|50.2!|50.3*|50.2_В|50.2_3|50.2_МИ|50.2_НА|50.2_ХИ|76.2_нач|50.2_СБ  
|50.2_СИ|50.2_У|50.2_Х|50.2_ДМИТ|50.2_Я|52!|54!|55!&(~55.3.1)|57.2!|63.2!|66.2!  
|67|67.2|68.2|68.3|68.4_2|68.4_3|68.5_2|68.5_3|70!&(~(70.1|70.1_РС|70.1_П|70.1_Ф  
|70.1_ФИЛ|70.1_С|70.1_Н))|71.2!|71.3!|73.2!|73.4!|75!&(~(75.ГР3.1|75.займ|75.ДОП  
|75.ГРЗ|75.УФ*!))|76.2!|78.РЕМ|75.ЗЕМ|82.2!|67.2!|75.УП!|78.3!|97.2!|96.2!  
|77.УП.СИК!|77.КМТ.СИК!';  
  ФильтрПараметров = 'суммавал<>руб';  
  КонечнаяДата = Now+1;  
  
  Разбиение[rdRow].ТипРазбиения =ByAcc;  
  Разбиение[rdTab].ТипРазбиения = byParam;  
  Разбиение[rdTab].Параметры = 'СуммаВал';  
  with ДобавитьПоказатель('СУММАВАЛ') do  
    Limit[skEndSaldo,roll]=true;  
    Limitvalue[skEndSaldo,roll]=0;  
    Limitkind[skEndSaldo,roll]=cutAboveAbs;  
  end;  
  Показыватьпрочие=false;  
  Build;  
  
  for j = 1..TableCount do  
    CurTable = j;  
    локВалюта=SplitValue[rdTab];  
    нач=локВалюта.Курс[Now-1];  
    кон=локВалюта.Курс[Now];  
    разница=(нач-кон);  
    if разница<>0:  
      for i = 1 .. RowCount do  
        CurRow = i;  
        CurColumn = 1;  
        -- if разница>0:  
        проводка баланс.91.3, SplitValue[rdRow] as Базовый,  
        сумма =(разница*unitvalue(EndSaldo(Roll,1)))^руб;
```

```

-- else
--     проводка баланс.91.3, SplitValue[rdRow] as Базовый,
--     сумма =(разница*unitvalue(EndSaldo(Roll,1)))^руб;
-- fi;

end;

end;

end;

end;

oper Проводка = "Просто проводка" (
    Сумма :unit спрВалюта;
    СД :Счет; ПД1 :String=""; ПД2 :String=""; ПД3 :String="";
    СК :Счет; ПК1 :String=""; ПК2 :String=""; ПК3 :String="";
    CRED :Numeric=0;
    Признаки :ParamStorage=nil;
    Комментарий :String=""
);
var Валюта :СпрВалюта;
var Сум :Numeric;
Сум = UnitValue(Сумма);
Валюта = UnitFactor(Сумма) as СпрВалюта;

if Валюта = Руб then
    if СД = Баланс.71.3 or СК = Баланс.71.3 then
        Проводка6( Сумма, Convert(Сум, Руб, Euro)^Euro, 0, СД, [ПД1, ПД2, ПД3],
            СК, [ПК1, ПК2, ПК3], [Признаки], Комментарий);
    else --if СД <> Баланс.71.3 and СК <> Баланс.71.3 then
        Проводка6( Сумма, Convert(Сум, Руб, USD)^USD, 0, СД, [ПД1, ПД2, ПД3], СК,
            [ПК1, ПК2, ПК3], [Признаки], Комментарий);
    end;
elseif Валюта = USD then
    Проводка6( Convert(Сум, USD, Руб)^Руб, Сумма, 0, СД, [ПД1, ПД2, ПД3], СК,
        [ПК1, ПК2, ПК3], [Признаки], Комментарий);
elseif Валюта = Euro then
    Проводка6( Convert(Сум, Euro, Руб)^Руб, Сумма, 0, СД, [ПД1, ПД2, ПД3], СК,
        [ПК1, ПК2, ПК3], [Признаки], Комментарий);
end;
end;

end

```

```
proc ВыполнитьЦиклПоВалюте(Sender: Object);
  var Q : ЗапросАналитики;
  var V : Валюта;
  var S : String;

  Q = ЗапросАналитики.Создать("Валюта");
  Q.OnlyUsed = False;
  Q.Select;

  while not Q.EOF do
    V = Q.Current;
    S = V.Name;
    if V.Mult then
      S = S + " * ";
    else
      S = S + " / ";
    end;
    S = S + Str(V.Rate[Today]) + " " + Str(V.Base);
    Trace(Str(V)+" : "+S+" -- "+V.Descr);
    Q.Next;
  end;
end;

proc ВыполнитьЦиклПоЕдИзм(Sender: Object);
  var Q : ЗапросАналитики;
  var V : ЕдИзм;
  Q = ЗапросАналитики.Создать("ЕдИзм");
  Q.Select;

  while not Q.EOF do
    V = Q.Current;
    Trace(Str(V)+" : "+V.Name+"-- "+V.Descr);
    Q.Next;
  end;
end;
```

Процедуры и функции служат для обработки информации, содержащейся в журналах, картотеках, бланках и файлах, а также предоставляют возможности по расчетам, формированию отчетных показателей и управлению прикладной программой.

В настоящем *Руководстве* для обозначения и процедур, и функций применяется обобщающий термин "метод".

Синтаксис описания процедур и функций приведен в темах:

[Заголовок процедуры и функции](#)

[Локальные переменные процедур и функций](#)

[Описание конца процедуры и функции](#)

[Описание вложенных процедур и функций](#)

Синтаксис вызова процедур и функций из других мест программы описывается в разделе [Оператор вызова процедуры или функции](#).

Процедуры начинаются с ключевого слова ПРОЦ (PROC). Функции начинаются с ключевого слова ФУНК (FUNC). После ПРОЦ или ФУНК должен следовать идентификатор метода (название процедуры или функции), а затем необязательный список аргументов, передаваемых в метод, заключенный в круглые скобки. Заголовок функции должен завершаться описанием типа возвращаемого функцией значения. Функция возвращает в качестве результата только одно значение определенного типа.

```
ПРОЦ <Идентификатор>
  [( <СписокПараметров> )];
ФУНК <Идентификатор>
  [( <СписокПараметров> )] : <Тип>;
```

Функции удобно использовать, например, в формулах переменных бланков. Поскольку функция возвращает значение определенного типа, она может быть одним из слагаемых или множителей в выражении или операндом сравнения. Пример использования функции в формуле:

```
ФУНК РасчетОстатка :Число;
  -- описание функции
  ...
КОНЕЦ;
Ост: Число = РасчетОстатка;
  -- переменная Ост вычисляется с помощью
  -- функции РасчетОстатка
```

Идентификатор метода не должен совпадать с идентификатором какой-либо переменной данного класса. По сути дела, и методы, и переменные являются равноправными свойствами класса и потому не могут "перекрывать" друг друга.

Метод может иметь так называемые формальные параметры (аргументы). Это набор специальных переменных, которые заполняются извне при вызове метода. Например, функция сложения двух целых чисел может иметь два аргумента целого типа, в которых будут находиться слагаемые. Сами же числа подставляются на место формальных параметров в тот момент, когда функция запускается на исполнение с помощью [оператор вызова](#).

Список аргументов указывается после имени процедуры или функции и заключается в круглые скобки. В списке аргументы разделяются друг от друга точкой с запятой, причем каждый аргумент описывается в виде

```
<ИмяАргумента> : <Тип>
```

Примеры описания заголовков процедур и функций:

```
ПРОЦ Выполнить; -- процедура без параметров
ФУНК СегодняшняяДата :Дата -- функция типа "Дата" без параметров
ПРОЦ Расчет (Слагаемое1 :Целое; Слагаемое2 :Целое)
  -- процедура с двумя целыми параметрами
ФУНК Проверка (Значение :Число) :Логическое
  -- функция логического типа с одним параметром числового типа
```

Для аргументов процедур и функций можно указать значение по умолчанию. Если таковое имеется, то соответствующий параметр при вызове метода может быть опущен. Синтаксис вызова процедур и функций из других мест программы описывается в разделе [Оператор вызова процедуры или функции](#). Синтаксис описания аргумента со значением по умолчанию следующий:

```
<ИмяАргумента> : <Тип> = <Выражение>
```

где **Выражение** должно иметь нужный тип. Значение выражения подставляется в качестве значения по умолчанию в отсутствие параметра для данного необязательного аргумента. В простейшем случае в качестве значения по умолчанию указывается константа.

Пример:

```
-- процедура принимает 2 необязательных параметра,
-- первый из которых по умолчанию будет считаться равным
-- случайному числу, а второй - нулю
proc TestDefVal (P1 :Integer = Random(100); P2 :Integer = 0);
  Trace(P1 + P2);
end;
```



```

proc Button3OnClick(Sender :Button);
  ClearTrace;
  TestDefVal(1, 1); -- оба параметра заданы
  TestDefVal(1); -- опущен второй параметр
  TestDefVal(,1);-- опущен первый параметр
  TestDefVal;    -- опущены оба параметра
end;

```

Аргумент метода может иметь два необязательных модификатора - **var (перем)** и **const (конст)**, которые пишутся перед именем аргумента:

```

(var|перем) <ИмяАргумента>:<Тип>
(const|конст) <ИмяАргумента>:<Тип>

```

Модификатор **var** означает, что аргумент используется для передачи значения из метода наружу (в вызывающий фрагмент кода). Если такого модификатора нет, то все изменения параметра внутри метода локальны, то есть при выходе из процедуры или функции теряются. Это удобно тем, что в отсутствии модификатора система гарантирует, что вызываемые методы не испортят содержимое передаваемых в них переменных. Однако, в тех случаях, когда аргумент действительно должен принимать измененное значение или получать новое значение из метода, его (аргумент) описывают с модификатором **var**.

Модификатор **const** означает, что аргумент не может быть изменен внутри метода. Компилятор запрещает использовать такие аргументы слева от знака присваивания или передавать их в качестве параметров другим процедурами и функциям, если только их соответствующий аргумент не описан также с модификатором **const**. Использование **const** позволяет приблизить стиль программирования к декларативному, когда компилятор, руководствуясь модификаторами, пресекает большую часть потенциальных ошибок.

Любая процедура или функция может иметь в дополнение к своему имени несколько синонимов, которые задаются в заголовке метода после ключевого слова **Синоним (Synonym)**. Синтаксис описания синонимов следующий:

```

ПРОЦ <Идентификатор>
  [ (Синоним|Synonym)
    <ДопИдентификатор_1> [ { ,<ДопИдентификатор_i> } ]
  ] [ <СписокПараметров> ];
ФУНК <Идентификатор>
  [ (Синоним|Synonym)
    <ДопИдентификатор_1> [ { ,<ДопИдентификатор_i> } ] ]
  [ <СписокПараметров> ] : <Тип>;

```

Данный синтаксис иллюстрируется [примером использования синонимов](#).

Локальные переменные процедур и функций

Для сохранения каких-то промежуточных значений, часто используемых при работе процедур и функций, возможно применение временных переменных. Такие переменные имеют ограниченное "время жизни", т.е. существуют только во время работы процедуры или функции, в которой они определены.

Это позволяет экономить ресурсы компьютера, его оперативную память, а также упростить структуру программы. Такие временные переменные называются локальными переменными процедуры.

Описание локальных переменных начинается с ключевого слова **Перем (Var)**. Далее следует перечисление идентификаторов переменных, разделенных запятыми, после чего через двоеточие указывается их общий тип.

В одной процедуре может быть несколько описаний локальных переменных. Все они должны помещаться между заголовком процедуры и первым ее оператором, т.е. локальные переменные должны быть описаны ранее, чем само тело процедуры.

Локальная переменная может совпадать по имени с [глобальной переменной](#). В этом случае она как бы "закрывает" доступ к последней. Поэтому говорят, что локальные переменные процедуры имеют приоритет перед глобальными переменными.

Доступ к локальным переменным возможен только внутри того метода, в котором они описаны. Обратиться к ним из других процедур даже того же самого класса или использовать в формулах переменных класса нельзя.

Это ограничение вызвано вполне понятными причинами: поскольку локальная переменная существует только ограниченное время пока выполняется соответствующая процедура, то может получиться, что к ней обратятся тогда, когда ее просто нет.

Примеры описания локальных переменных:

```
Перем Сумма1, Сумма2 :Число;  
Перем Подходит :Логическое;
```

В языке ТБ.Скрипт существует возможность создания вложенных процедур и функций. Правила использования вложенных методов такие же, как и в языке Паскаль:

- вложенный метод виден только внутри основного (охватывающего) метода, т.е. обращаться к нему можно из самого метода, во всех вложенных методах, описанных ниже внутри основного метода, а также непосредственно в теле основного метода;
- во вложенном методе доступны: основной метод, локальные переменные основного метода, описанные выше данного вложенного метода, и параметры основного метода;
- возможно несколько уровней вложенности.

Вложенные процедуры удобно использовать, например, в том случае, когда нужно выполнить одну и ту же последовательность действий несколько раз внутри тела одной процедуры/функции, но больше нигде в составе этого класса ее делать не требуется. Тогда не имеет смысла создавать отдельный метод класса (объекта), так как он используется только внутри одного метода.

В [иерархии классов](#) методы, содержащие вложенные процедуры и функции показываются в виде дерева.

Пример 1:

```
-- основная процедура
proc SimpleTest;
  var I :Integer;

  -- вложенная процедура, использующая
  -- доступ к локальным переменным основной процедуры
  proc LocProc1(Index :Integer);
    -- здесь доступны Self и I
    -- J - недоступна, так как описана ниже
    Trace(Str(Self) + ' ' + Str(I*Index));
    I = Random(100);
    Trace(I);
  end;

  var J :Integer;

  I = Random(100);
  Trace(I);
  for J=1..100 do
    LocProc1(J);
  end;
end;
```

Пример 2:

```
-- Вычисление факториала разными способами
proc FactorialTest;
-- внешняя процедура

-- вложенная функция для расчета первым способом
-- уровень вложенности 1
func Fact1 (N :Integer) :Numeric;
-- локальная процедура
-- уровень вложенности 2
proc LocFact(N :Integer; var Res :Numeric);
var r :Numeric;
  if N > 0 then
    -- рекурсивный вызов себя
    LocFact(N-1, r);
    Res = r * N;
  else
    Res = 1;
  end;
end;
-- вызов локальной функции
```

```

    LocFact(N, Result);
end;

-- вложенная функция для расчета вторым способом
-- уровень вложенности 1
func Fact2 (N :Integer) :Numeric;
    -- локальная процедура, для которой используется
    -- то же имя, что и в первой функции, но конфликта имен
    -- нет, так как Fact1.LocFact отсюда не видна
    -- уровень вложенности 2
    proc LocFact(N :Integer; var Res :Numeric);
        if N > 0 then
            -- вызов родительской функции
            Res = Fact2(N - 1) * N;
        else
            Res = 1;
        end;
    end;
    LocFact(N, Result);
end;

-- вложенная функция для расчета третьим способом
func Fact3 (N :Integer) :Numeric;
    if N > 0 then
        Result = Fact3(N - 1) * N;
    else
        Result = 1;
    end;
end;

-- вложенная функция для расчета четвертым способом
func Fact (N :Integer) :Numeric;
var I :Integer;
    Result = 1;
    for I=2..N do
        Result = Result*I;
    end;
end;

-- начинается тело внешней процедуры

var f, f1, f2, f3 :Numeric;
var I :Integer;

for I=0..100 do
    f = Fact(I);
    f1 = Fact1(I);
    f2 = Fact2(I);
    f3 = fact3(I);
    -- Проверяем, что результат всех функций
    -- совпадает с точностью до Eps
    Assert( (f - f1) * (f - f1) < Eps );
    Assert( (f1 - f2)* (f1 - f2) < Eps );
    Assert( (f2 - f3)* (f2 - f3) < Eps );
    Assert( (f3 - f) * (f3 - f) < Eps );
    Trace('Factorial(' + Str(I) + ') = ' + Str(f));
end;
end; -- конец внешней процедуры

```

Описание конца процедуры и функции

Для окончания процедуры или функции можно использовать ключевое слово КОНЕЦ (END), используемое повсюду в ТБ.Скрипте для завершения описаний, либо инвертированные (написанные "задом наперед") ключевые слова, открывавшие описание процедуры или функции - соответственно ЦОРП или CORP для процедуры и КНУФ или CNUF для функции.

Использовать ли для окончания процедуры слово КОНЕЦ или инверсию открывавшего описание слова - дело вкуса программиста. Использование во всех случаях слова КОНЕЦ придает единообразие текстам; с другой стороны, инвертированные слова позволяют компилятору языка ТБ.Скрипт производить более точный контроль правильности написанных текстов.

Описание процедуры или функции должно заканчиваться точкой с запятой.

Учет - это, прежде всего, вычисления. Порядок и содержание вычислений во всех прикладных проектах задается с помощью выражений. Выражение - это выполненная по специальным правилам формализованная запись, состоящая из чисел, строк, переменных и операций над ними.

Выражения используются в журналах в качестве сумм операций и проводок, значений параметров типовых хозяйственных операций, в формулах для вычислений в бланках, в фильтрах картотек, в калькуляторе.

Например, чтобы записать проводку на сумму налога на добавленную стоимость, можно, конечно, предварительно посчитать НДС на калькуляторе, а затем подставить вычисленное значение в проводку. Однако значительно удобнее записать это в виде выражения, вычисляющего необходимый процент от суммы операции. В этом случае, во-первых, исключается всякое ручное вычисление; во-вторых, и это самое главное - данное выражение будет автоматически пересчитываться каждый раз, когда это необходимо.

Язык выражений, как и любой другой, имеет собственные правила записи, называемые синтаксисом. Эти правила одинаковы для всех прикладных проектов и могут отличаться лишь некоторыми аспектами использования.

Более подробные сведения можно получить в темах:

[Формальное описание выражений](#)

[Приоритет операций](#)

[Идентификаторы в выражениях](#)

[Использование комментариев](#)

[Массивы](#)

[Функции в выражениях](#)

Идентификаторы в выражениях используются для адресации к объектам, которые данные идентификаторы обозначают. Так, по идентификатору [переменной](#) программа получает хранящееся в ней значение, а встретив в выражении идентификатор [функции](#) - вычисляет эту функцию и подставляет в выражение ее значение.

Напомним, что в Студии [идентификатор](#) может состоять из латинских и русских букв, цифр и знаков подчеркивания. Идентификатор не может начинаться с цифры, поскольку это не позволит программе отличить его от числа.

```
$Идентификатор = (Буква | СимволПодчеркивания)  
[ { (Буква | Цифра | СимволПодчеркивания) } ]
```

Идентификатор может быть квалифицированным, т.е. состоять из нескольких простых идентификаторов, разделенных точками.

```
$КвалифицированныйИдентификатор = Идентификатор  
[ { .Идентификатор } ]
```

Обычно квалифицированные идентификаторы используются для обозначения переменных, у которых есть нечто общее. В этом случае они различаются последним идентификатором, входящим в состав квалифицированного.

Например, счета одного и того же плана счетов имеют квалифицированные имена, состоящие из имени плана и, через точку, имени счета, например, "Баланс.01" и "Баланс.02". Это позволяет группировать однотипные сущности (в данном случае - счета) по функциональному назначению и упрощает понимание сути прикладного проекта (особенно, если это большой проект).

Особую роль полностью квалифицированные идентификаторы играют в связи с тем, что встроенный в Студию язык ТБ.Скрипт является [объектно-ориентированным](#) и потому практически все переменные, константы, процедуры и функции входят в состав того или иного класса. При этом обращение к членам одного класса из другого класса, как правило, осуществляется с помощью полного имени, включающего не только имя переменной (или метода, то есть функции или процедуры класса), но и имя самого класса.

Например, если в [прикладном проекте](#) Студии определен класс **A** с переменной **Сумма**, то для доступа к ней из класса B нужно написать идентификатор **A.Сумма**.

Составные идентификаторы используются также и непосредственно в иерархии классов. Например, все типы накладных, необходимые в прикладном проекте, имеет смысл описать в виде единой группы классов, содержащей классы конкретных типов, в частности, товарно-транспортной накладной. Сама группа накладных также может входить в другую более глобальную группу, охватывающую всю "первичку". Тогда каждый класс документов получит квалифицированное имя, состоящее из последовательно состыкованных названий групп (начиная с самой крупной), к которым относится данный класс, и простого идентификатора класса. Так, продолжая пример с накладными, мы получим документ:

Первичка.Накладные.ТТН

Чем сложнее программа, тем тяжелее в ней разобраться. Поэтому в Студии допустимо чередование выражений с *комментариями* - строками произвольного текста, которые не влияют на логику работы программ.

Признаком начала комментария являются два минуса. Начиная от них, вся строка до конца считается пояснением, и при компиляции программы игнорируется. Поэтому в комментариях можно писать любой поясняющий текст.

Примеры комментариев:

```
-- далее идет вычисление оборотов  
x = 2+3; -- эта сумма всегда равна 5
```

В первом примере комментарием является вся строка, во втором - ее часть, выделенная курсивом.

Массивы

В выражениях наравне с обычными переменными могут использоваться многозначные переменные, называемые массивами. Массив - это несколько однотипных значений, объединенных одним именем. Доступ к элементам массива осуществляется по его имени и порядковому номеру требуемого элемента.

Например, если в программе определен числовой массив Цены, то для того чтобы получить или задать в нем третью цену необходимо написать "Цены[3]".

Подробная информация об использовании массивов приводится в разделе, посвященном [переменным-массивам](#).

В Студии приняты стандартные правила приоритетности операций, которые используются в математике.

Наибольшим приоритетом обладают операции типа умножения: * (умножение), % (процент от числа), / (деление). Все операции этого типа имеют одинаковый приоритет.

Затем идут операции типа сложения: + (сложение чисел или конкатенация строк), - (вычитание). Сложение и вычитание также имеют одинаковый приоритет.

Наконец, наименьшим приоритетом обладают логические операции. Среди них первой выполнятся операция НЕ, затем И, а самыми низкоприоритетными считаются операции ИЛИ и исключающее ИЛИ.

В случае, если необходимо изменить порядок вычислений, следует, как и в математике, использовать круглые скобки. Например, в выражении

$2+3*4+5$

первой выполнится умножение тройки на четверку. Однако, записав

$(2+3)*(4+5)$,

программист обеспечивает более высокий приоритет операциям сложения и тем самым перемножает уже их результаты.

Хорошим стилем при программировании считается использование круглых скобок в любых более или менее сложных выражениях. Это позволяет повысить понятность математической записи и избежать неочевидных и трудноуловимых ошибок.

Рекомендуется при записи выражений использовать скобки для явного задания порядка их вычисления.

Особо отметим, что выражения целого типа везде, где это необходимо, автоматически преобразуются к числовому типу. Однако размер ячеек памяти компьютера у целых чисел значительно меньше, чем у чисел с дробной частью, поэтому когда результат выражения не должен быть целым, рекомендуется указывать "0" в дробной части у первого из используемых чисел. Например, вместо

$1'234'567'890 + 1'234'567'890$

надо писать

$1'234'567'890.0 + 1'234'567'890$

Рассмотренные выше понятия - [типы данных](#), [константы](#) и [переменные](#) разных типов - как правило, используются в Студии не по отдельности, а в составе выражений, будучи объединенными какими-либо операциями. При этом для того, чтобы выражение было правильным и могло быть вычислено, необходимо, чтобы указанные в нем операции были допустимы для значений данного типа.

Выражение состоит из операндов и операторов. Оператор обозначает выполняемое над операндами действие. Операнд фактически представляет собой часть исходного выражения, выступающую как единое целое для оператора.

Операторы бывают бинарными и унарными. Бинарные операторы выполняют действие над двумя операндами, которые записываются слева и справа от оператора. Унарные операторы выполняют действие над одним операндом и записываются перед ним.

Операторы имеют различный [приоритет](#).

Синтаксические диаграммы для выражений Студии выглядят следующим образом:

```
$Выражение          = [ '(' ]
                      ( Операнд БинарныйОператор Операнд
                      | УнарныйОператор Операнд
                      | Операнд )
                      [ ')' ]
$Операнд             = Константа
                      | Переменная
                      | ВызовФункции
                      | Выражение
$БинарныйОператор    = АрифметическийОператор
                      | ЛогическийОператор
                      | ОперацияСравнения
$АрифметическийОператор = '*' | '/' | '%' | '+' | '-'
$ЛогическийОператор   = ( 'AND' | 'И' ) | ( 'OR' | 'ИЛИ' )
$ОперацияСравнения    = '>' | '<' | '=' | '<>' | '<=' | '>='
$УнарныйОператор      = АрифметическийУнарный
                      | ЛогическийУнарный
$АрифметическийУнарный = '+' | '-'
$ЛогическийУнарный     = ( 'NOT' | 'НЕ' )
```

Арифметические бинарные операторы '+' и '-' применимы к операндам всех типов кроме логического. Арифметические бинарные операторы '*', '/' и '%' применимы к операндам всех типов кроме логического, строкового и даты. Арифметические унарные операторы '+' и '-' применимы к операндам всех типов кроме строкового, логического и даты. Логические операторы применимы только к операндам логического типа. Операции сравнения должны выполняться над операндами одного и того же типа, а результат выполнения таких операций имеет логический тип.

Как видно из приведенной выше схемы, выражение может включать в себя арифметические, строковые и логические операции. При этом результат одного выражения может служить операндом для другого.

Например, вы можете сложить два числа и проверить, не равен ли результат третьему. В этом случае результат операции сложения становится участником операции сравнения.

Для того, чтобы правильно определить порядок выполнения операций в сложном выражении, необходимо задать правила старшинства среди операций, т.е. их [приоритет](#).

Функции в выражениях

В состав выражений, как правило, входят не только переменные и константы, над которыми выполняются элементарные операции, но и функции, выполняющие более сложную обработку, объединяя в себе десятки и сотни операций.

Например, если в программе часто возникает необходимость подсчитать сумму чисел в различных массивах, то имеет смысл написать функцию, вычисляющую сумму элементов любого массива, а потом вызывать ее из выражений.

Синтаксис описания функций рассматривается в одном из [последующих разделов](#). Здесь же мы лишь затронем вопрос использования функций в составе выражений. Функции записываются в составе выражений по аналогии с переменными, однако, в отличие от последних, функции могут иметь список параметров (заключенный в круглые скобки), которые передаются внутрь функции. Если функция не имеет аргументов, то она используется в составе выражений точно так же, как переменная.

Например, в выражении "ДатаОтчета-Сегодня" используется переменная **ДатаОтчета** и функция **Сегодня**. Оба слагаемых имеют тип **Дата**, для которого определена операция вычитания.

Некоторые наиболее часто проводимые вычисления, в частности те, что встречаются в повседневных бухгалтерских задачах, реализованы в Студии в составе [встроенных программных классов ТБ.Скрипт](#). Входящие в них функции применяются в выражениях наравне с пользовательскими функциями, определенными в прикладных проектах Студии.

Например, при построении различных отчетов нужно уметь получать суммы оборотов по заданным счетам. Поэтому многие из таких базисных вычислений просто включены в Студию. Их наличие позволяет сделать процесс программирования более простым и удобным, а сами программы - компактными, быстродействующими и безошибочными.

Студия поддерживает несколько стандартных типов данных. Тип данных определяет внешнее и внутреннее представление элемента данных (например, переменной или параметра функции) и то, как его можно использовать.

Например, денежная сумма - это число, причем действительное число (способное иметь дробную часть), а наименование фирмы - это строка. Очевидно, что сумму можно делить, умножать и выполнять над ней другие арифметические действия, а вот со строкой такие операции недопустимы.

Каждый тип данных в Студии имеет несколько обозначений (вариантов записи). Ниже приведен перечень простых (элементарных) типов:

Целое, Целый, Integer	Целый
Число, Numeric, Real	Числовой
Логическое, Логический, Logical	Логический
Строка, String	Строка
Дата, Date	Дата

ТБ.Скрипт допускает неявное приведение типа Integer к Numeric, поэтому, например, вызов функции

```
Сло ( Выражение:Число ):Строка;
```

можно осуществить не только с параметром типа Numeric, но и Integer. Это относится к любому фрагменту кода, где используются данные типы. Обратное приведение некорректно, то есть присвоить значение типа Numeric параметрам или переменным типа Integer нельзя. По аналогии логический тип можно привести к числу.

Кроме того, в Студии используются объектные типы данных, образуемые встроенными или пользовательскими классами. Вот примеры нескольких встроенных классов:

Класс	Сущность
Запись, Record, Документ, Document	Запись
Объект, Object	Объект
Отчет, Report	Отчет
Запрос, Query	Запрос

Простые и объектные типы равноправны с точки зрения их использования в функциях, процедурах и выражениях.

Объектные типы данных подробно рассматриваются при описании соответствующих (одноименных) классов.

Существует еще один тип, который играет весьма важную роль в программировании на ТБ.Скрипт, - это [Вариант|Variant](#). Переменные и аргументы типа вариант могут содержать значение любого из вышеперечисленных типов.

Для расширения перечня стандартных типов Студии имеется возможность в каждом классе дополнительно определить так называемые пользовательские типы.

Кроме того, в ТБ.Скрипт существует возможность описать переменную, параметр или константу (включая и массивы), которая бы содержала не объект класса, а ссылку на сам класс. Например, при создании запросов к информационной базе необходимо указывать перечень классов записей, по которым нужно выполнять запрос. В этом случае используется массив классов. Более подробно описание типов "ссылка на класс" приводится в темах:

- [Описание переменных.](#)
- [Целое / Integer](#)
- [Число / Numeric](#)
- [Логическое / Logical](#)
- [Строка / String](#)
- [Дата / Date](#)
- [Вариант / Variant](#)
- [Объектные типы](#)
- [Приведение типов](#)
- [Пользовательские типы](#)
- [Специальные типы](#)

Тип **Вариант** предназначен для работы со значениями произвольного по своей сути содержания. Это может быть и число, и дата, и объект. **Вариант** является универсальным типом, к которому можно привести любой другой тип, но не наоборот.

Если переменной типа **Вариант** была присвоена, например, дата, то к этой переменной можно применять лишь операции, свойственные датам, а ее использование в выражениях или в качестве параметра функции, где требует другой тип, недопустимо. В таких случаях на стадии исполнения программа будет выдавать сообщение об ошибке.

Тип "Дата" предназначен для работы со значениями временных отсчетов, включающих в себя день, месяц, год, часы, минуты и секунды. Год может быть задан двумя или четырьмя цифрами. Допустимые обозначения типа: **Дата, Date.**

Синтаксис записи константы типа "Дата" можно определить следующим образом. Вначале через точку записываются число, месяц и год, причем число и месяц могут быть представлены целыми положительными числами как с одним, так и с двумя разрядами в зависимости от конкретной даты (например, первое января 2000 может быть записано как 01.01.00 или как 1.1.00). Год также допускается указывать в полной или краткой форме, то есть с двумя разрядами или четырьмя (например, 01.01.00 или 01.01.2000). Затем после пробела может идти временная часть даты: часы, минуты и секунды записываются в виде чисел с одним или двумя разрядами, причем числа эти отделены друг от друга двоеточием (например, "20:30:15" - половина девятого и 15 секунд). Секунды или минуты вместе с секундами могут быть опущены, тогда они считаются равным нулю.

Программа проверяет значения типа "Дата" на корректность.

Примеры дат:

21.08.91 26.7.1965

Примеры дат со временем:

05.01.2002 18:00:00 05.01.2002 18:00

Над датами возможны следующие операции:

- Сложение с целым числом (дней)
- Вычитание целого числа (дней)
- Вычитание даты из даты (с точностью до дня)

Все эти операции выполняются только над той частью переменной, в которой хранятся год, месяц и число. Остальная часть переменной, связанная со временем (часы, минуты, секунды), этими операциями не используется.

При необходимости можно воспользоваться [функциями](#) работы с датами.

Например, сложение даты с числом "19.08.97+2" даст "21.08.97", т.е. через 2 дня после 19 августа наступит 21 августа.

Из даты можно вычесть целое число, например, "12.12.97-12" даст 30.11.97, т.е. за 12 дней до 12 декабря было 30 ноября.

Из даты можно вычесть дату, например, "21.08.1991-7.11.1917" даст "26950", т.е. от 7 ноября 1917 года до 21 августа 1991 года прошло 26950 дней.

Над значениями типа "Дата" можно производить операции сравнения:

<	раньше
>	позже
=	одновременно (с точностью до дня)
<=	раньше или одновременно
>=	позже или одновременно
<>	не одновременно

Операции сравнения дают значения логического типа (ИСТИНА или ЛОЖЬ).

Примеры использования операций сравнения:

"12.12.97 > 1.12.97" даст ИСТИНА

"13.12.90 > 1.11.90" даст ЛОЖЬ

Логический тип служит для выражения одного из двух возможных значений: ИСТИНА или ЛОЖЬ.

Наиболее близким аналогом логического типа данных в реальной жизни является электрический выключатель. Он также может быть либо включен, либо выключен, и третьего не дано.

Логический тип описывается с помощью зарезервированных слов **Логическое**, **Логический**, **Logical**. Возможные значения данного типа (логические константы) обозначаются идентификаторами ИСТИНА (TRUE) и ЛОЖЬ (FALSE).

Несмотря на кажущуюся простоту и ограниченность, логический тип очень широко применяется при программировании. В частности, все операции сравнения дают результат именно логического типа. Действительно, утверждение о том, что "сальдо равно нулю" может быть либо верным (ИСТИНА), либо ошибочным (ЛОЖЬ).

Над переменными логического типа допустимы все операции сравнения. При сравнении двух логических величин на "больше" или "меньше" считается, что ЛОЖЬ всегда меньше, чем ИСТИНА.

Кроме того, для значений логического типа в ТБ Скрипте введены логические операции:

- логического умножения И (AND);
- логического сложения ИЛИ (OR);
- исключающего ИЛИ (XOR);
- отрицания НЕ (NOT).

Все они также возвращают результат логического типа. Суть каждой операции поясняют следующие таблицы:

Операция ИЛИ

Операнд 1	Операнд 2	Результат
ИСТИНА	ИСТИНА	ИСТИНА
ИСТИНА	ЛОЖЬ	ИСТИНА
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ

Операция И

Операнд 1	Операнд 2	Результат
ИСТИНА	ИСТИНА	ИСТИНА
ИСТИНА	ЛОЖЬ	ЛОЖЬ
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ

Операция XOR

Операнд 1	Операнд 2	Результат
ИСТИНА	ИСТИНА	ЛОЖЬ
ИСТИНА	ЛОЖЬ	ИСТИНА
ЛОЖЬ	ЛОЖЬ	ЛОЖЬ

Операция НЕ

Операнд	Результат
ИСТИНА	ЛОЖЬ
ЛОЖЬ	ИСТИНА

Таким образом, операция ИЛИ дает значение ИСТИНА, если по крайней мере один из операндов истинен; операция И - если истинное значение имеет и тот, и другой операнд; операция XOR дает значение ИСТИНА, когда только один из операндов имеет значение ИСТИНА; операция НЕ применяется к одному операнду и возвращает обратное ему значение (такую операцию называют еще инвертированием).

Объектные типы

Поскольку программа построена на основе объектно-ориентированных принципов, в ней на внутреннем уровне поддерживается большое число объектных типов, которые представляют собой классы. К их числу, например, относятся [встроенные классы](#), такие как **Запись (Документ)**, **Отчет**, **Шаблон**. Прикладной программист может создавать свои классы, производные от того или иного встроенного класса.

Для объектных типов определены следующие операции:

= присваивания
.
 разыменования

Первая из них присваивает ссылку на объект, хранящуюся в одной переменной объектного типа, другой переменной того же типа. Сам объект при этом не дублируется. Вторая операция предназначена для обращения к свойствам объекта (полям, процедурам, функциям). Синтаксис записи следующий:

<объектная переменная>.<свойство объекта>

Например, для установки начальной и конечной дат при построении отчета можно записать:

```
Отчет1.ДатаНач = Дата1;  
Отчет1.ДатаКон = Сегодня;
```

Здесь:

Отчет1 - переменная объектного типа [Отчет](#);
ДатаНач и **ДатаКон** - свойства класса [Отчет](#);
Дата1 - переменная типа "Дата";
Сегодня - функция класса [Система](#), возвращающая текущую дату.

Операции разыменования могут быть последовательными, если свойство объекта в свою очередь является объектом. Например, корректна следующая запись:

```
БланкНакладной.Шаблон.ТекущаяКлетка.Шрифт.Жирный = ИСТИНА;
```

Здесь изменяется шрифт в текущей клетке некоего бланка накладной.

Более подробно о сути объектных типов рассказывается в разделе [Введение в объектно-ориентированное программирование](#).

В дополнение к стандартным типам ТБ.Скрипт у программиста существует возможность определить собственные типы, которые называются обобщающим термином *пользовательские* и могут происходить как от стандартных типов, так и от других пользовательских.

Синтаксис описания пользовательского типа следующий:

```
type <Идентификатор типа> = <Тип>;
```

Таким описанием вводится пользовательский тип с указанным идентификатором и содержанием, детализируемым выражением справа от знака равенства. Пример:

```
type RecordClasses = Class [] Record;
```

Пользовательский тип принадлежит тому классу, в котором он описан. В этом смысле тип можно рассматривать как особого рода свойство класса. Выражаясь программистским языком, идентификатор пользовательского типа определен в пространстве имен класса. Таким образом, идентификатор типа не может совпадать ни с другим идентификатором типа, описанным в этом же классе, ни с именем члена этого класса (объекта). Пользовательские типы отображаются в браузере классов (после компиляции) наравне с остальными свойствами и методами классов.

Полностью квалифицированный идентификатор типа, по которому можно обращаться к пользовательскому типу из других классов проекта (и подпроектов), составляется из имени проекта, в котором определен класс с пользовательским типом, имени этого класса и собственно идентификатора типа:

```
<Имя проекта класса>.<Имя класса типа>.<Идентификатор типа>
```

Одной из разновидностей пользовательского типа является перечисляемый тип, определяемый как набор поименованных целочисленных констант.

Синтаксис:

```
type <Идентификатор типа> =  
  (<Идентификатор> [ = <Инициализация> ]  
  [ {, <Идентификатор> [ = <Инициализация> } ]  
  );
```

После идентификатора типа, за знаком равенства, следует заключенная в круглые скобки последовательность идентификаторов (они отделяются друг от друга запятыми), каждый из которых обозначает один из элементов перечисления. Для каждого элемента может быть указана необязательная целочисленная константа, задающая его значение.

Элементы перечисления интерпретируются как свойства класса, в котором описан тип, причем эти свойства - суть целочисленные константы. Их значения либо берутся из части <Инициализация> для конкретного элемента (если она есть), либо определяются автоматически. В том случае, если для какого-либо элемента перечисления часть <Инициализация> отсутствует, его значение будет на единицу больше предыдущего или же равно 1, если этот элемент стоит первым в списке.

Перечисляемый пользовательский тип является типизированным целым типом, в связи с чем в выражениях требуется явное приведение из одного типа в другой: как из Integer в перечисление, так и наоборот.

Пример:

```
public  
  type TVarType =  
    (varUnknown = 0, -- неизвестно что (0)  
    varString, -- строка (1)  
    varInt, -- 4-байтовое целое (2)  
    varNumeric, -- число с плавающей точкой (3)  
    varLogical, -- логическое значение (4)  
    varDate, -- ДатаВремя в формате TDateTime (5)  
    varObject, -- объект (6)  
    varArray = 8, -- массив (8)  
    varClass, -- указатель на класс (9)  
    varNull -- пусто (10)  
    );  
private  
  FTypeNames :String[] =  
    ["Неизвестный тип",  
    "Строка",  
    "Целое",  
    "Число",  
    "Логическое",  
    "Дата",
```

```
"Объект",  
"лишний элемент!!!",  
"Массив",  
"Класс",  
"Пусто"]];  
public
```

```
func NameOfType(AType :TVarType) :String;  
  Result = FTypeNames[Integer(AType)];  
end;
```

```
func VarTypeStr(Value :Variant) :String;  
  Result = NameOfType(TVarType(VarType(Value)));  
end;
```

В данном примере определяется перечисляемый тип TVarType. В комментариях в круглых скобках указано значение, соответствующее каждому элементу перечисления. Следует отметить, что нумерация начата с 0, а не с 1, как было бы по умолчанию, а значение 7 намеренно пропущено.

В функциях NameOfType и VarTypeStr выполняется явное приведение типов в обе стороны.

В рамках Студии часто используются так называемые преобразования формата, позволяющие задать с помощью последовательности управляющих символов внешнее представление данных, то есть то, как они должны отображаться на экране. В частности, преобразования формата используются в шаблонах бланков и в некоторых функциях системы, например, в функции Str (Str).

Стандартный формат числа предусматривает использовать в качестве разделителя целой и дробной части точку (".") и кроме того допускает использование апострофа в качестве разделителя троек разрядов (триад).

Формат числа определяется следующими управляющими символами:

- . - позиция десятичной точки;
- # - позиция для одного десятичного знака (незначащие нули не отображаются)
- \$ - позиция для одного шестнадцатеричного знака (незначащие нули не отображаются)
- 0 - позиция для одного знака (незначащие нули отображаются как "0"); может использоваться совместно как с #, так и \$
- , - признак необходимости разделения триад (позиция размещения запятой в формате не играет роли)

Для отображения целых достаточно указать 0, для действительных чисел полезно задать количество знаков после десятичной точки, например, 0.00. Данный формат показывает, что после запятой будет отображаться два знака, а перед запятой будет выведено столько знаков, сколько требуется для полного отображения всего числа. При необходимости дробная часть округляется в большую или меньшую сторону в соответствии с общепринятыми правилами (если первый отбрасываемый разряд больше или равен 5, то последний из сохраняемых разрядов увеличивается на 1).

Различие между символами '#' и '0' легко понять из следующего примера. Если, скажем, имеется число 123.4567, то по формату 0000.00, оно будет выведено как 0123.46 (обратите внимание на незначащий ноль, добавленный впереди), а по формату #####.## - 123.46. Важно отметить, что количество управляющих символов перед точкой не влияет на реальный размер получаемой строки, поскольку он зависит от самого числа. То есть, целая часть числа 123456.783 при использовании формата #####.## никак не будет обрезана - результат 123456.78.

Возможно одновременное задание различных форматов для отображения положительных, отрицательных чисел и нуля - для этого форматы разделяются символом ";", например:

```
Деб=+,0.00;Кре=,0.0;-
```

Данный формат иллюстрирует применение сразу нескольких приемов работы. Здесь через точку с запятой указаны три разных формата, первый из которых будем использован для положительных чисел, второй - для отрицательных, третий - для нуля. В форматной строке допускается указывать произвольные неуправляющие символы (в данном случае "Деб=+", "Кре="). Явно записанный знак "+" всегда будет присутствовать у положительных чисел. Запятые диктуют программе разделять тройки десятичных разрядов апострофами.

Студия допускает применение других, отличных от стандартных разделителей. Такие символы задаются в конце форматной строки в квадратных скобках:

- Gx - символ "x" используется вместо стандартного разделителя триад
- Dx - символ "x" используется вместо десятичной точки

Например, для заданного формата ,0.00[D=G_] число 12345.6 отображается следующим образом: 12_345=60.

Для значений типа "Дата" применяются следующие форматы отображения:

- d - день месяца без ведущего нуля;
- dd - день месяца с ведущим нулем;
- m - номер месяца без ведущего нуля;
- mm - номер месяца с ведущим нулем;
- mmm - краткое название месяца (январь, фев....);
- mmmm - полное название месяца (январь, февраль);
- yy - две последние цифры года;
- yyyy - полное представление года;
- ddd - краткое название дня недели (пн., вт....);
- dddd - полное название дня недели (понедельник, вторник....);
- hh - часы;
- mm - минуты;
- ss - секунды.

Пример: "dd.mm.yyyy hh:mm:ss".

При использовании различных типов в программах часто возникает необходимость преобразовать один тип к другому. Такое преобразование называют *приведением типов*. Некоторые случаи приведения типов выполняются системой неявно, то есть, образно выражаясь, незаметно для программиста. Так, целое (**Integer**) может быть автоматически преобразовано компилятором в десятичное число (**Numeric**). Обратное преобразование невозможно, в силу наличия дробной части числа, которую пришлось бы отбросить. Компилятор специально не выполняет таких преобразований, позволяя обнаружить потенциальные ошибки. Однако, если программисту действительно необходимо привести десятичное число к целому, он может сделать это с помощью соответствующих функций, тем самым явно задав требуемое приведение. Например, для приведения типа **Numeric** к **Integer** следует воспользоваться функцией **Int** на самом деле, это метод класса [Система](#)). Функция **Int** принимает один параметр типа **Variant** и возвращает соответствующее значение типа **Integer**. Другие функции, могущие быть полезными для преобразования типов: **СтрокаВДату**, **Стр**, **КакЧисло**.

Особый случай приведения типов заключается в преобразовании одного [объектного типа](#) к другому. В этом случае приведение возможно либо от производного типа к одному из родительских, например, от типа **Кнопка** к типу **ОбъектШаблона** (такое приведение выполняется компилятором неявно), либо наоборот - от базового класса к одному из классов-наследников, причем компилятор проверяет, действительно ли ссылка на базовый объект хранит экземпляр производного класса, и если это условие не выполняется - генерирует ошибку. Для приведения объектных типов используется запись:

```
<выражение целевого типа> = ИмяКласса (<выражение исходного типа>);
```

где ИмяКласса - это название класса целевого типа.

Так, продолжая пример с кнопкой и объектом шаблона, можно записать:

```
ОбъектШаблона1: ОбъектШаблона;  
Кнопка1: Кнопка; -- кнопка на шаблоне бланка  
Кнопка2: Кнопка; -- кнопка на шаблоне бланка  
Запрос1: Запрос;  
-- ...  
  
-- выполняется неявное приведение  
ОбъектШаблона1 = Кнопка1;  
-- ...  
-- явное приведение  
Кнопка2 = Кнопка(ОбъектШаблона1);  
  
-- явное некорректное приведение  
Кнопка1 = Кнопка(Запрос1); -- ошибка при компиляции
```

Здесь переменная ОбъектШаблона1 типа **ОбъектШаблона** сначала получает значение, равное ссылке на объект Кнопка1, затем Кнопка2 получает значение из переменной ОбъектШаблона1 - это допустимо, так как переменная базового класса содержит объект требуемого класса **Кнопка**. По аналогии, можно привести переменную базового класса к значению любого класса, лежащего в иерархии на пути наследования между базовым классом и классом того значения, которое фактически хранится в переменной.

Приведение класса Запрос к классу Кнопка невозможно, так как они относятся к разным веткам иерархии (соответственно, Объект->Запрос и Объект->ОбъектШаблона->Кнопка).

Другой пример:

```
БазоваяФорма: Форма;  
Форма1: ФормаБланка;  
Форма2: ФормаКартотеки; <  
-- ...  
Форма = Форма1; -- Форма содержит "бланк"  
-- ...  
Форма2 = ФормаКартотеки(Форма); -- ошибка при компиляции:  
-- попытка привести "бланк" к "картотеке"
```

Приведение объектных типов позволяет реализовать в программе на языке ТБ.Скрипт принципы [полиморфизма](#). Полиморфизм - это специальный механизм оперирования различными по своей сути объектами с помощью единообразных методов. Например, переменная типа **ОбъектШаблона** может быть использована для хранения объекта любого производного типа, включая кнопки, поля ввода, флаги, переключатели и другие. При этом, вне зависимости от того, какой именно объект содержится в переменной, с помощью универсального набора методов (процедур и функций), определенных для класса **ОбъектШаблона**, можно определить местоположение объекта на шаблоне бланка, его размеры, строку с подсказкой, надпись, а также управлять видимостью объекта, его выводом на печать, делать его недоступным или, наоборот, активизировать.

Некоторые базовые неobjектные типы являются совместимыми по внутреннему формату хранения и потому могут приводиться друг к другу с помощью такой же конструкции, что и объектные, однако вместо имени класса в ней указывается имя типа. Например, типы **Логическое**, **Целое**, а также **Вариант**, хранящий значение одного из этих двух типов, допускается преобразовывать друг в друга. Аналогично можно работать и с [пользовательскими типами](#), если они совместимы с типом **Целое** (например, перечисление).

Приведение типов также может осуществляться с помощью специального оператора **as (как)**. Общий синтаксис следующий:

<выражение целевого типа> = <выражение исходного типа> as <целевой тип>;

Здесь в качестве типа может выступать не только стандартный тип ТБ.Скрипта, но и объектные типы.

Строковый тип представляет собой различные комбинации символов, т.е. слова, словосочетания и целые предложения. В строке могут присутствовать алфавитно-цифровые символы, символы пунктуации и широкий набор специальных печатаемых символов (например, ® © §). Специальные непечатаемые (управляющие) символы, например, символ абзаца, в строке недопустимы.

Допустимые обозначения типа: **Строка, String**. Вся текстовая информация в программе имеет строковый тип.

Для обозначения константы строкового типа используется последовательность символов, записанная в кавычках или апострофах.

Строка может быть ограничена как одинарными, так и двойными кавычками. При этом внутри кавычек не могут встречаться одинарные кавычки того типа, который используется для ограничения строки. Строка должна заканчиваться тем же символом-ограничителем, которым была начата.

Использование двух видов ограничителей строки при необходимости позволяет использовать в строках и одинарные, и двойные кавычки. Для этого достаточно лишь использовать другой вид кавычек в качестве ограничителя, например, " АОЗТ 'Аврора' " или ' АОЗТ "Аврора" '. Кроме того, для обозначения символа кавычки внутри строки можно использовать удвоенный символ кавычки, например, " АОЗТ ""Аврора"" " (с двойными кавычками) или ' АОЗТ "Аврора" ' (с одинарными кавычками).

Над строками можно производить операции сравнения, а также специальную операцию конкатенации ("склеивания"), обозначаемую знаком "плюс".

Примеры строковых констант и операций над строками:

"Баланс по форме" + " 1"

-- результат данной операции - строка "Баланс по форме 1".

"А"<"Б"

-- результат данной операции - ИСТИНА, поскольку в таблице

-- символов Windows буква А идет раньше буквы Б, т.е. в этом

-- смысле буква Б "больше", чем буква А

Целое / Integer

Целый тип данных используется для хранения чисел, которые не имеют дробной части. Он находит широкое применение, например, для задания номеров документов. Допустимые обозначения: **Целое, Целый, Integer**. Значением целого типа является целое число, то есть положительное или отрицательное число без дробной части (нет ни десятичной точки, ни цифр после нее).

В целых числах допускается применять специальные символы-разделители, отделяющие произвольное число разрядов друг от друга. В качестве разделителей могут использоваться символ подчеркивания ("_") и апостроф ("'").

Положительным считается как число со знаком "+" впереди, так и без этого знака (неявная запись). Отрицательные числа всегда должны содержать в начале знак "минус". Диапазон значений [-2¹⁴⁷·483⁶⁴⁸... +2¹⁴⁷·483⁶⁴⁷].

Примеры значений целого типа (целочисленных констант):

17 -5 1'000'000 2_500 +45

Над целыми числами допустимы следующие арифметические операции:

+	сложение
*	умножение
-	вычитание
/	деление
%	взятие процента от числа

а также операции сравнения:

<	меньше
>	больше
=	равно
<=	меньше или равно
>=	больше или равно
<>	не равно

Результатом операций сравнения являются значения логического типа - ИСТИНА или ЛОЖЬ.

Операции сравнения, перечисленные выше, допустимы почти для всех стандартных типов Студии.

Числовой тип используется для тех чисел, которые могут содержать дробную часть, т.е. для действительных чисел. Например, суммы денег, включающие копейки, являются действительными числами. Допустимые обозначения объектов числового типа - **Число, Numeric, Real**.

Числовой тип имеют все операции деления, даже если и делимое, и делитель - целые числа, поскольку при операции деления всегда возможен остаток, и, следовательно, дробная часть.

В действительных числах допускается применять те же специальные символы-разделители (для выделения разрядов), что и в целых числах, то есть подчеркивание ("_") и апостроф ("'").

Положительным считается как число со знаком "+" впереди, так и без этого знака (неявная запись). Отрицательные числа всегда должны содержать в начале знак "минус". Диапазон значений - приблизительно 1.1×10^{4932} .

Примеры действительных чисел:

17.0 -5.3509 1'000.68 2_500.50

Для данных числового типа определены те же операции, что и для [целых чисел](#) т.е. арифметические операции и операции сравнения.

ТБ.Скрипт позволяет получать информацию о типах объектов во время исполнения проекта. Такая информация, называемая по-английски run-time type information (RTTI), позволяет узнавать для любой объектной переменной или параметра имя и характеристики этой сущности. В число характеристик входят имя класса, количество и перечень членов класса, включая их область видимости (личный - публичный), признак того, является ли класс встроенным в ядро или реализованным на ТБ.Скрипт, а также все прочие характеристики, заложенные в двоичный код программы на основе компиляции исходных текстов.

Если какая-либо переменная (или параметр) не является объектной сущностью, информация о ней всё равно может быть получена. Однако в этом случае сущность, разумеется, не будет иметь членов, так как она не является составной как объект класса или класс.

Следует понимать, что информация о типах, описывающая классы, методы, свойства и т.д., предоставляется в свою очередь через объекты специальных классов с набором методов и свойств для извлечения этой информации.

Базовым классом среди классов для получения RTTI выступает класс ИнфЧлена / MemberInfo. Его наследниками в иерархии классов Студии являются ИнфКласса / ClassInfo, ИнфПеречисления / EnumTypeInfo, ИнфПрограммногоТипа / UserTypeInfo, ИнфМетода / MethodInfo, ИнфСобытия / EventMethodInfo, ИнфПоля / FieldInfo. Назовем их для общности RTTI-классами.

Один объект RTTI-класса (производного от **ИнфЧлена**), описывает одно свойство, метод, или определение пользовательского типа в некотором классе. Если какое-либо свойство в свою очередь является объектным, то по нему можно получить набор объектов с RTTI, описывающий члены этого класса, и так далее. Аналогичным образом можно анализировать объектные параметры методов. При получении информации о типе может оказаться, что он в свою очередь базируется на другом типе - ссылку на него также можно получить с помощью свойств RTTI-классов. RTTI-классы предоставляют полный набор инструментов для "расшифровки" внутренней структуры данных.

Особый (нетипичный) случай представляет собой получение RTTI-информации об объектах самих RTTI-классов - например, ничто не мешает получить описание класса **ИнфКласса**, которое будет возвращено в виде объекта того же класса и фактически описывать себя. Это вряд ли окажется полезным на практике, однако позволяет вникнуть в суть работы RTTI-классов и относиться к ним, как к обычным классам Студии.

Необходимость в использовании RTTI возникает в том случае, если реализуемый алгоритм должен быть достаточно абстрактным, не привязанным к типам (классам), известным на стадии компиляции проекта. Например, есть задача вывода в DBF-файл записей произвольных классов, причем в проекте эти классы не известны (такая ситуация, в частности, существует, если все абстрактные функции используют абстрактные базовые классы и вынесены в подпроект, а вызываются из других проектов). В этом случае единственным способом узнать, какие поля содержит класс записи, является чтение RTTI.

При написании текстов на встроенных языках программирования - ТБ.Скрипт, МТЛ, язык описания структуры учета - разработчик имеет возможность пользоваться специальными сервисными функциями редактора текстов.

Прежде всего, это - автоматическая подсветка синтаксиса. Ее настройка осуществляется в диалоге настройки редактора текстов, на [странице "Цвета"](#). По умолчанию, подсветка выделяет ключевые (служебные) слова, а также помечает различным цветом латинские и русские буквы в идентификаторах.

Текстовый редактор тесно связан с [отладчиком](#) языка ТБ.Скрипт. Практически все команды отладчика (например, установка контрольной точки или пошаговое выполнение) доступны непосредственно из редактора, когда в нем открыт отлаживаемый модуль. Состояние отладчика (точки останова, текущая выполняемая строка кода, метки строк, для которых был сгенерирован двоичный код) отображается в текстовом редакторе.

Если окно [иерархии классов](#) вызывается из редактора, программа предварительно просматривает внутренние структуры данных, сгенерированные в процессе компиляции COD-файлов, в поисках имени файла, открытого в данный момент в редакторе, и идентификатора, на котором находится текстовый курсор. Если соответствие будет найдено, программа не только откроет окно с иерархией классов, но и установит в нем курсор на строку, в которой зарегистрирована указанная в редакторе сущность. (Например, если текстовый курсор в редакторе подведен к идентификатору метода Икс класса Игрек, то после вызова окна иерархии классов в нем будет подсвечен именно этот метод.) Если соответствие не будет найдено, окно иерархии открывается вне всякого контекста - активной становится вершина иерархии.

Дополнительно в программе имеется специальная команда **Перейти к описанию**. Когда курсор в редакторе находится на некоторой лексеме (идентификаторе класса, метода, свойства или же имени типа или константе), данная команда позволяет открыть COD-файл, в котором находится описание указанной сущности, и перейти в нем непосредственно к тому месту, где сущность описывается. Иными словами, из строки с кодом, использующей возможности некоторого класса, можно контекстно перейти в код этого класса.

Если курсор в редакторе установлен на сущность не пользовательского, а встроенного класса, то открывается окно справки для данной сущности. Важно, что переход к описанию возможен только из строк, генерирующих код. Так, любые идентификаторы в комментариях не могут быть "расшифрованы" с помощью данной команды.

Для быстрого вызова команды **Перейти к описанию** достаточно нажать клавишу **Ctrl** и, удерживая ее, подвести курсор мыши к требуемому фрагменту кода - при этом фрагмент подсвечивается контрастным цветом и подчеркиванием, если по нему действительно есть дополнительная информация, - а затем выполнить щелчок левой кнопкой мыши. В результате открывается окно "Ссылки", содержащее список, в котором перечислены имена файлов с исходным текстом и номерами строк в них, где используется выделенный элемент (класс, метод, свойство и т.д.). Двойной щелчок на нужной строке списка позволяет перейти в нужное место исходного текста.

Следует иметь в виду, что проверка имен сущностей выполняется контекстно, в полном соответствии с семантикой использования конкретного фрагмента строки (идентификатора). Например, если в коде записана строка:

```
Chart1.SeriesKind = График.График;
```

то переход для первого слова "График" осуществляется на справку по классу **График**, а переход для второго слова "График" - на справку по константам, задающим виды графиков.

В редакторе существует множество команд, которыми удобно пользоваться при программировании (см. разделы с перечнем команд Редактор (правка) и Редактор (блоки)). В частности, это команда включения/отключения режима автовыравнивания строк: если режим включен, то в новой строке (по нажатию клавиши **Enter**) курсор по умолчанию устанавливается в позицию, с которой начинаются символы в предыдущей строке. Особо следует отметить команду **Закомментировать** (Ctrl+'-'). Она позволяет установить или снять комментарий сразу с группы строк, выделенных блоком.

В целях облегчения программирования, а именно - для упрощения рутинного набора стандартных синтаксических конструкций языков ТБ.Скрипт, МТЛ и других, текстовый редактор программы предоставляет режим автоматического формирования текста по предварительно подготовленным заготовкам. Настройка режима автоформирования выполняется на странице "Формирование" диалога настройки редактора.

Принцип работы автоформирования (при условии, что поименованные заготовки соответствующим образом подготовлены в вышеупомянутом диалоге) следующий. При редактировании некоторого файла достаточно ввести название заготовки и по нажатию пробела (по умолчанию), вместо названия заготовки в файл автоматически подставляется весь её текст. Если использование пробела в качестве инициатора автоподстановки отключено в диалоге настроек, то выполнить автоформирование можно с помощью команды Автоформирование текста.

Автоформирование текста работает только в файлах с расширениями *.cod, *.mtl, *.lis, *.jur.

По умолчанию система поставляется с большим набором стандартных заготовок, облегчающих набор программ на языках ТБ.Скрипт, MTL, языке описания структуры учета.

Разработка бланков, как и других объектов прикладных проектов в целом, ведется с применением современного объектно-ориентированного подхода. В рамках этой концепции каждый бланк является объектом некоторого прикладного класса, производного от класса [ФормаБланка](#), языка [ТБ.Скрипт](#) и задается с помощью взаимосвязанной пары файлов: cod-файла с исходным кодом, задающим логику (алгоритм) работы бланка, и tpl-файла с шаблоном (экранной формой) бланка. Внешний вид бланка создается с помощью [визуального редактора шаблонов](#).

По сути дела бланк – это совокупность объектов, относящихся к абстрактным классам стандартной иерархии классов, на основе которых с помощью языка ТБ.Скрипт создаются производные пользовательские классы бланков, и в частности электронные аналоги первичных, служебных и отчетных документов произвольной сложности. Поскольку бланки – это всего лишь специализированный фрагмент иерархии классов языка ТБ.Скрипт, их разработка во многом аналогична написанию [классов языка ТБ.Скрипт](#). Все принципы программирования на языке ТБ.Скрипт, синтаксис и семантика операторов остаются едиными. Самое главное отличие бланка от "чистого" класса, который имеет лишь алгоритмическое описание в COD-файле, – это наличие дополнительной визуальной формы (шаблона) в TPL-файле.

Встроенный объектно-ориентированный язык программирования ТБ.Скрипт позволяет управлять поведением бланков программно. С его помощью можно описывать произвольные окна, способные выдавать запросы к пользователю, получать от него ответы, обрабатывать полученную информацию, отображать результаты расчетов и влиять на логику работы программы.

Специфические особенности разработки бланков изложены в темах:

- [Модели бланков](#)
- [Регистрация бланков в проекте](#)
- [Файл описания бланка](#)
- [Использование объектов на шаблоне](#)
- [Процедуры и функции в бланках](#)
- [Наследование бланков](#)

Кроме секций шаблон может содержать и находящиеся в бланке объекты, такие как кнопка, флаг или переключатель. Каждый из этих объектов выполняет определенную функцию, например, по нажатию кнопки можно выполнить то или иное действие. Часто бывает нужно задать объектам какое-то особенное поведение: в нужный момент изменять надписи на кнопках или заполнять секцию. Все это позволяет делать встроенный в Студию язык программирования ТБ.Скрипт.

Например, бланк содержит информацию о сотруднике. Нажатие на кнопку позволяет показать либо краткую биографическую справку, либо полную. В этом случае для удобства использования необходимо, чтобы в тот момент, когда на экране представлена краткая информация, на кнопке было бы написано *"Показать полную справку"*, а когда в бланк выведена полная биографическая справка, надпись на кнопке должна быть изменена на *"Показать краткую справку"*.

Выполнение подобных действий над объектами также осуществляется с помощью переменных бланков. Для каждого объекта, которому необходимо задать особенности в его поведении, в бланке заводится соответствующая переменная. Она как бы указывает на этот объект, и все обращения к ней переадресуются ему. С помощью данной переменной можно сделать запрос к объекту, например, выяснить, какой именно рисунок виден на экране в данный момент, а также изменить внешний вид или свойства объекта: сменить надпись на кнопке, загрузить другой рисунок и т.д.

Так, в следующем примере в COD-файле некоторого бланка описана переменная Label1 класса **Надпись (Label)**. Предполагается, что одноименный объект соответствующего типа был добавлен на шаблон того же бланка. При изменении состояния переключателя, который также присутствует на шаблоне, текст надписи динамически изменяется.

```
Label1: Label;  
proc ПриИзменении(R1: RadioButton);  
  -- в зависимости от состояния переключателя  
  if (R1.Name = "InButton") AND (R1.State = TRUE) then  
    Label1.Caption = "Приходная накладная:";  
  else  
    Label1.Caption = "Расходная накладная:";  
  end;  
end;
```

Более подробно применение объектов на шаблоне рассматривается в разделе [Объекты](#).

Моделью бланка называется способ его описания на внутреннем языке Студии для задания внешнего вида и функциональных свойств бланка. Под этим термином понимают всю совокупность средств и приемов, позволяющих создать эффективно работающий бланк. Разработка бланков в Студия выполняется с помощью *модели бланки на шаблонах*.

Так называемая *традиционная модель* в программе не используются, хотя и является достаточно простым и эффективным способом описания бланков. Бланки этой модели полностью описывались в одном текстовом файле. Однако она не позволяет полноценно использовать некоторые возможности среды Windows.

В программе поддерживается работа только с бланками на шаблонах. Название этой модели полностью передает ее основную особенность: внешний вид формируемого бланка создается не с помощью языковых средств, а в виде специального шаблона, создаваемого встроенным визуальным редактором шаблонов бланков. Шаблон по своему внешнему виду напоминает электронную таблицу, в нем допустимо использование многих полезных и красивых возможностей пользовательского интерфейса системы Windows, а сам процесс разработки новых бланков при этом значительно облегчается и становится более понятным.

Программа реализована с применением объектно-ориентированного подхода, и бланк является ни чем иным как объектом некоторого прикладного класса, производного от класса [ФормаБланка](#), который задается с помощью пары файлов: COD-файла с исходным кодом, задающим логику работы бланка, и TPL-файла с шаблоном (экранной формой) бланка. Для получения более подробной информации см. раздел [Разработка бланков](#) и [Язык ТБ.Скрипт](#).

Бланки являются полноценными классами Студии и поддерживают механизмы наследования и полиморфизма, описанные в разделе [Наследование и полиморфизм](#).

Однако существуют и некоторые ограничения. Так нельзя сделать бланк производным от пользовательского класса ("чистого" класса без шаблона) и, наоборот, пользовательский класс - от бланка. То же самое верно и для пары классов бланк - картотека. Иными словами, генеральное свойство бланка - тот факт, что он является бланком, - наследуется наравне с программно определенными свойствами. Поэтому бланк нельзя унаследовать от класса, не обладающего таким генеральным свойством.

К вышеизложенным правилам наследования при наследовании бланков-редакторов добавляются следующие дополнительные ограничения:

- бланк-редактор может быть унаследован от другого бланка (редактора или не редактора);
- обычный бланк не может быть унаследован от бланка-редактора, то есть должен быть унаследован от другого обычного бланка;
- если бланк-редактор унаследован от другого бланка-редактора, то он должен редактировать запись либо того же класса, что и базовый, либо унаследованную от этой записи;
- шаблон бланка не наследуется в том смысле, что его нельзя дополнять в производном классе бланка, определив там дополнительные интерфейсные элементы, однако в случае отсутствия шаблона у производного бланка будет использоваться шаблон предка. Кроме того для получения бланка с набором секций родительского бланка существует возможность вызова специальной процедуры **Форма.ЗагрузитьШаблон**, которая принимает два параметра: первый - имя подгружаемого шаблона (таким образом, это может быть как родительский, так и любой другой шаблон), и второй - целочисленный аргумент, позволяющий управлять местом появления подгружаемых секций в текущем шаблоне. Главное ограничение при этом в том, что объекты подгружаемого шаблона игнорируются (не грузятся). Таким образом, если ни один из вышеперечисленных приемов не удовлетворяет требованиям разработчика, необходимо скопировать tpl-файл базового бланка под именем производного бланка и отредактировать как необходимо.

В переменных бланка хранится информация, а с помощью шаблона задается форма ее отображения на экране и при печати. Кроме этого, возможности бланка позволяют также производить обработку введенной первичной информации и получать отчетные и обобщенные данные.

Для описания действий, которые необходимо произвести над введенной в бланк информацией, служат процедуры и функции бланка, принципы использования которых изложены в разделе [Процедуры и функции](#).

Все бланки, используемые в работе на том или ином участке автоматизированного учета, должны быть зарегистрированы в проекте, соответствующем этому участку. Осуществляется это с помощью команды **Добавить** (Ins) контекстного меню из [окна редактора проекта](#). При этом курсор должен быть установлен на объекте "Формы бланков".

Когда в иерархии объектов выделен определенный бланк, в этом же меню доступны команды **Удалить** (Del) и **Переименовать** (Enter), а также команда **Добавить папку** (Alt+Ins). С помощью них можно удалить бланк из проекта, изменить его название или создать папку для хранения группы бланков.

В результате выполнения команды **Добавить** открывается многостраничный диалог "[Создание бланка](#)", который под руководством Мастера позволяет быстро и безошибочно создать бланк.

Пользователи Студии могут изменить бланк (например, добавить в него новые переменные или скорректировать алгоритм того или иного расчета) самостоятельно только в том случае, если ими была приобретена лицензия, включающая доступ в режим разработки. При этом они получают возможность изменить исходный текст бланка. Однако, следует иметь в виду, что формы документов, используемые в деятельности бухгалтерии, часто изменяются. Чтобы облегчить пользователям решение возникающих в связи с этим проблем, разработчики программы проводят постоянный контроль соответствующих нормативных актов и регулярно обновляют комплекты бланков при каждом изменении законодательства. Изменения, сделанные пользователями в оригинальных бланках проектов, не позволят выполнить их обновление - изменения будут утеряны и их потребуются повторить.

Вышеперечисленных проблем можно избежать, если предварительно сделать резервную копию оригинального бланка, созданного фирмой разработчиком программы или иной фирмой, поставляющей бланки. В дальнейшем, при необходимости можно будет вернуться к оригинальному бланку.

Например, в составе стандартной поставки имеется бланк *.cod, который требуется подкорректировать. Следует скопировать данный бланк в другое место диска (причем вместе с соответствующим файлом *.tpl") или на дискету, изменить содержание бланка и перекомпилировать проект.

Описание бланка в файле начинается с заголовка, в котором задается название документа и некоторые его параметры. Далее описываются переменные бланков, а также процедуры и функции, задающие алгоритмы изменения и взаимосвязей используемых переменных. Описания переменных и процедур могут перемежаться.

Ключевые слова при описании бланков записываются русскими или латинскими буквами. Список ключевых слов ТБ.Скрипт приведен в [Приложении](#).

Файл описания бланка является типичным модулем с исходным текстом на [языке ТБ.Скрипт](#) в котором в процессе разработки бланков могут использоваться следующие конструкции языка:

- [Заголовок бланка](#)
- [Наименование бланка](#)
- [Атрибут бланка-редактора картотеки](#)
- [Список синонимов](#)

Заголовок бланка

Вид заголовка бланка схематично изображен на следующей диаграмме:

```
Бланк "<ИмяКлассаБланка>"
[ , Редактор <ИмяКлассаРедактируемойЗаписи> ]
[ Синоним
  <ДопИмя_1> [ { , <ДопИмя_i> } ] ];
```

или в англоязычной нотации

```
Class "<BlankClassName>"
[ , editor <ИмяКлассаРедактируемойЗаписи> ]
[ Synonym
  <NikName_1> [ { , <NikName_i> } ] ];
```

Заголовок бланка начинается с ключевого слова **Бланк** (англоязычный синоним **Blank**) или **Класс (Class)**. Рекомендуется записывать его с первой позиции строки. После этого должно стоять наименование бланка, заключенное в кавычки. Затем может следовать атрибут бланка-редактора, за которым, при необходимости, допускается указать синонимы класса. В конце заголовка должна стоять точка с запятой.

Примеры простых заголовков (без атрибута и синонимов):

```
Бланк "Платежное поручение";
Blank "Работник";
```

Иногда требуется, чтобы заголовок бланка отличался от того, что выводится на закладке окна бланка. Чаще всего такая необходимость возникает, если заголовок довольно длинный, и в то же время желательно сэкономить место на панели с закладками. В подобных случаях имеется возможность задать разные строки для заголовка и закладки. Синтаксис описания следующий:

```
Бланк "<Заголовок окна>|<Текст на закладке>";
```

Т.е. в заголовке описания бланка через вертикальную черту указываются две строки.

Например:

```
Класс "Товарно-транспортная накладная|ТТН",
редактор Документы.Накладная;
```

Наименование бланка

Наименование бланка представляет собой строковую константу, т.е. строку, заключенную в одинарные или двойные кавычки.

Под заданным таким образом наименованием бланк отображается в списке доступных бланков. Оно высвечивается также в заголовке окна бланка.

Как правило, наименование содержит название документа с точки зрения прикладной области автоматизации, например, бухгалтерии. Так, для бланка ПОРУЧП - это "Платежное поручение".

Атрибут бланка-редактора картотеки

Атрибут **Редактор (Editor)** объявляет описываемый бланк бланком-редактором для той или иной записи (структуры данных). За ключевым словом **Редактор** следует имя класса документов, которые должен

редактировать бланк.

Подробнее назначение и функционирование бланков-редакторов будет рассмотрено при описании [редактирования картотеки](#).

Пример заголовка бланка-редактора:

Бланк "Карточка товара", **Редактор** ТОВАРЫ;

Список синонимов

Список синонимов позволяет назначить дополнительные имена для класса бланка. Задается список с помощью ключевого слова **Синоним (Synonym)**, после которого через запятую перечисляются все требуемые синонимы. Список должен располагаться в конце заголовка класса (бланка), непосредственно перед точкой с запятой.

Пример заголовка бланка-редактора с двумя синонимами:

Бланк "Платежное поручение", редактор Реестр.Поручения Синоним Платежка, Поручение;

Определенные, таким образом, синонимы можно использовать везде, где должно было бы стоять имя соответствующего класса.

Пример заголовка бланка-редактора с двумя синонимами:

Бланк "Платежное поручение", редактор Реестр.Поручения Синоним Платежка, Поручение;

Картотеки предназначены для визуального отображения одного или нескольких классов записей в специальном [окне картотеки](#). Разработка картотек прикладным программистом ведется в окне редактора проекта с применением стандартного для программы объектно-ориентированного подхода. В рамках такого подхода картотека рассматривается как набор объектов нескольких классов, в которых описаны ее свойства и модель поведения.

С помощью объектно-ориентированного языка ТБ.Скрипт прикладной программист имеет возможность расширить функционал стандартных классов картотек и увязать их в единое целое с другими сущностями – бланками, пользовательскими классами, интерфейсом программы. Процесс разработки включает в себя формирование трех взаимозависимых файлов с описанием алгоритмической и визуальной (внешнего представления) частей, а также еще и файла с настройками картотеки, который создается автоматически.

Алгоритмическая часть картотек определяется с помощью языка ТБ.Скрипт в COD-файле, имеющем то же имя, что и картотека. Картотеки являются полноценными [классами](#) языка ТБ.Скрипт, наравне с пользовательскими и системными классами, а также бланками. По аналогии с бланками, картотека может иметь шаблон (TPL-файл), который позволяет в случае необходимости ввести в окно картотеки какие-либо дополнительные управляющие элементы (кнопки, флаги, поля ввода) и тем самым, расширить функционал стандартного [окна картотеки](#).

Принципы работы с COD-файлами и шаблонами подробно описаны в разделах, посвященных программированию на [языке ТБ.Скрипт](#), и [редактору шаблонов](#). Принципы разработки шаблонов бланков и картотек практически не отличаются.

Особенности написания COD-файлов для картотек в основном обуславливаются необходимостью использовать системный класс [Картотека / CardFile](#) и связанные с ним классы ([СтолбецКартотеки](#), [ФормаКартотеки](#) и т.д.). Все эти классы подробно рассмотрены в разделе, посвященном [иерархии классов](#).

Наличествующий у картотеки специальный BRO-файл хранит интерфейсные настройки картотеки, такие как названия и ширина столбцов, их порядок, параметры отображения записей и пр. BRO-файл редактируются самой системой, по мере того как разработчик манипулирует шаблоном картотеки в визуальном редакторе бланков. Некоторые из настроек, хранимых в BRO-файле, (например, порядок колонок) относятся к так называемым пользовательским настройкам, то есть задаются конечным пользователем в процессе эксплуатации прикладного проекта и хранятся на локальном компьютере.

Картотека, унаследованная от другого родительского класса картотеки, может не иметь BRO-файла – в этом случае он будет браться из родительской картотеки или из любого другого наиболее близкого предка, в котором имеется BRO-файл. То же самое относится и к TPL-файлам шаблонов.

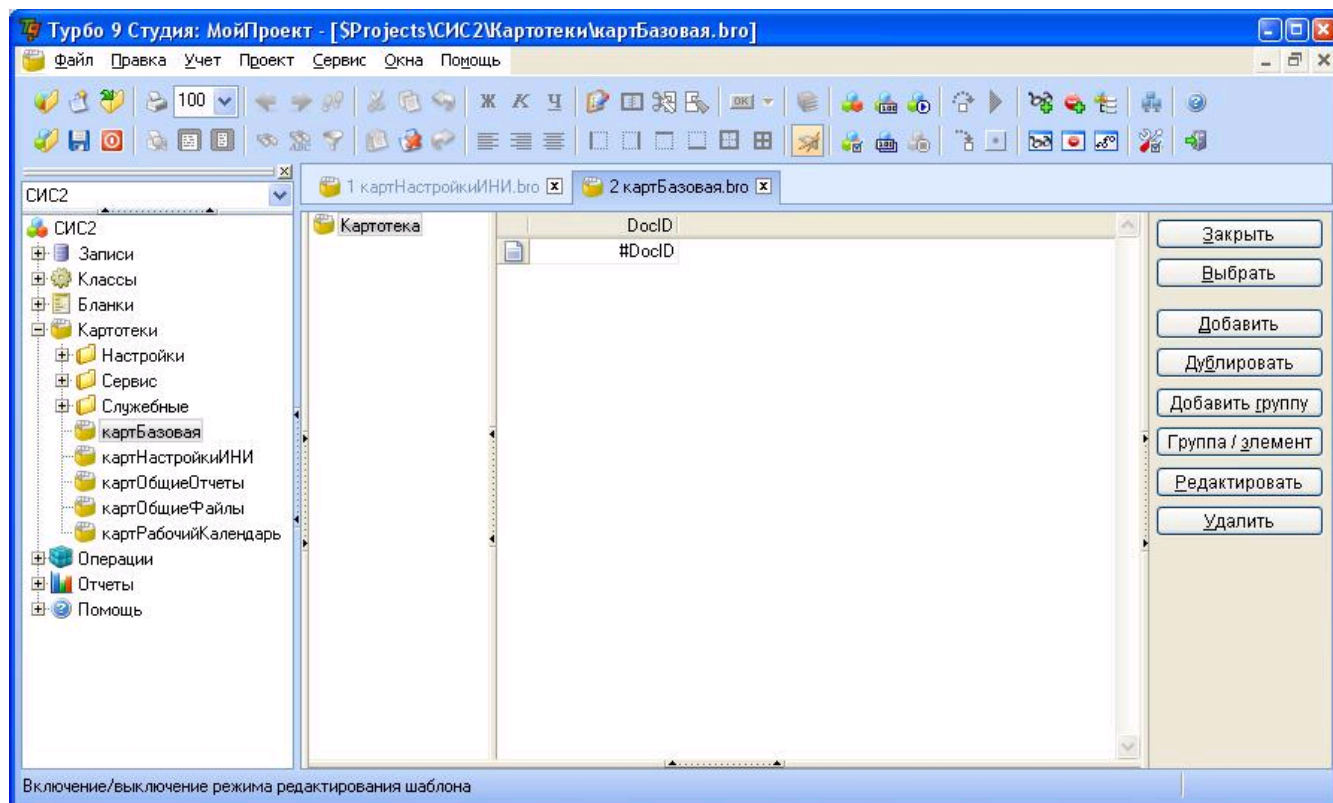


Рис. Разработка картотек.

К особому виду картотек относятся журналы-картотеки и картотеки-аналитики. Свойства [журналы-картотек](#) и [картотеки-аналитики](#) описываются на встроенном языке в файлах структуры учета.

Более подробно вопросы разработки и настройки картотек отражены в темах:

- [Создание и регистрация картотек в проекте](#)
- [Использование бланка для редактирования картотеки](#)
- [Управление картотеками из процедур и функций](#)

- [Картотеки как журналы операций](#)
- [Картотеки как аналитические справочники](#)
- [Взаимодействие картотек и бланков](#)
- [Настройка картотек](#)
 - [Настройка свойств картотеки](#)
 - [Общие свойства картотеки](#)
 - [Добавление, удаление и размещение подтаблиц](#)
 - [Свойства подтаблицы](#)
 - [Свойства столбцов](#)
 - [Кванторы](#)
 - [Перечень функций, используемых в фильтрах](#)
 - [Служебные поля записей в фильтрах](#)

Пользователь может настраивать внешний вид картотеки под свои нужды, перемещая колонки и управляя их видимостью, включая или отключая иерархическое представление и т.д. Еще большей свободой в выборе средств отображения и логики поведения картотеки наделен прикладной разработчик Студии. В частности он может добавлять в картотеку вычисляемые поля (их значения берутся не из базы данных, а вычисляются "на лету" на основе значений других полей), накладывая на нее фильтр, искусственно сужая область видимости записей, изменять внешний вид самого окна картотеки, добавляя в него дополнительные элементы управления (кнопки, флаги и т.д., а также вставлять картотеки во фреймы бланков и использовать фреймы в самих картотеках.

Состав записей картотеки может изменяться с помощью [фильтра](#). Положение и размер столбцов настраивается с помощью мыши, а их состав – в диалоге ["Настройка видимости колонок"](#). Колонки можно менять местами методом "перетаски и отпусти", то есть достаточно подвести указатель мыши к колонке, которую следует переместить, нажать левую кнопку мыши и, не отпуская ее, перетянуть колонку на новое место.

Ширина колонки также регулируется с помощью мыши. Когда указатель мыши помещен над границей между колонками в области шапки таблицы, пользователь получает возможность "подцепить" границу и двигать ее влево или вправо. Если выполнить на границе колонок двойной щелчок, то находящаяся слева колонка получает атрибут автоматического вычисления ширины. Ширина всех колонок с автоподбором ширины вычисляется как частное от деления общего размера окна картотеки (за вычетом колонок с точно заданной шириной) и числа автоподстраиваемых колонок. Автоподбор ширины колонки отключается, как только пользователь вновь изменит ширину колонки.

Общие сведения по настройке и фильтрации картотек приведены в темах:

- [Настройка свойств картотеки](#)
- [Общие свойства картотеки](#)
- [Добавление, удаление и размещение подтаблиц](#)
- [Свойства подтаблицы](#)
- [Свойства столбцов](#)
 - [Кванторы](#)
 - [Перечень функций, используемых в фильтрах](#)
 - [Служебные поля записей в фильтрах](#)

Если класс записи содержит многозначные поля и структуры, их можно отображать в виде подтаблиц в специальной области окна картотеки. Эта область по умолчанию располагается в нижней части окна и отделена от табличной части подвижной перемычкой. В этой области доступны команды контекстного меню, предназначенные для работы с подтаблицами.

Добавление подтаблицы выполняется командой **Добавить подтаблицу**. При этом вызывается диалог "Выберите подтаблицу", где разработчик может выбрать из списка одно из имеющихся многозначных полей или структур записи.

Для гетерогенных картотек, созданных на основе нескольких классов записей, в диалоге "Выберите подтаблицу" перечислены только те структурные многозначные поля записей, которые имеют одинаковые имена и типы во всех используемых классах записей.

Для удаления подтаблицы необходимо щелкнуть по ней в окне картотеки, вызвать контекстное меню и выполнить в нем команду **Удалить подтаблицу**.

Когда в окно картотеки вставлено несколько подтаблиц, они могут быть по-разному размещены относительно друг друга. В частности, подтаблицы могут быть размещены на отдельных закладках, либо делить область подтаблиц между собой по горизонтали или вертикали. Способ их расположения определяется с помощью команд контекстного меню, которое вызывается либо для перемены между таблицами (если они в данный момент делят область подтаблиц между собой), либо для закладок (если каждая подтаблица в данный момент выводится на собственной странице с закладкой). Меню содержит команды: **По горизонтали**, **По вертикали**, **С закладками**. Слева от пункта, который соответствует выбранному в данный момент режиму, выводится флажок.

Квантор – это специальная синтаксическая конструкция, допустимая только внутри выражений фильтров и используемая для наложения условий на содержимое подтаблиц картотеки, то есть структурных многозначных полей. Ниже подробно рассматриваются все кванторы.

Существует | Exists

Этот квантор позволяет проверить истинность некоторого логического выражения для записей в подтаблице структурного поля и возвращает значение ИСТИНА, если хотя бы одна из записей в подтаблице удовлетворяет указанному условию, которое должно содержать имена полей подтаблицы.

Синтаксис:

<СтруктурноеПоле>. Exists | Существует (<выражение>)

где

<СтруктурноеПоле> – имя структурного поля картотеки;
<выражение> – произвольное выражение логического типа, в котором могут использоваться имена полей структурного поля.

Например, пусть в проекте существует MTL-описание следующего документа со структурным полем:

```
Document Doc1;  
  Field F1 :integer;  
  Struct StructArray1 array integer;  
    Field F21 :integer;  
    Field F22 :string;  
  end;  
end;
```

Данный класс документов будет использоваться и во всех остальных примерах данного раздела. Тогда можно написать следующий фильтр:

```
"F1 > 0 and StructArray1.Exists(F21 = 0 and F22 = 'услуга')"
```

Здесь условие налагается как на содержимое поля F1 основной таблицы (записи Doc1), так и полей подтаблицы StructArray. В отфильтрованную часть картотеки попадут те документы, в которых F1 больше 0 и в подтаблице есть хотя бы одна строка, где F21 равно нулю, а F22 равно литералу 'услуга'.

Все / All

Этот квантор позволяет проверить истинность некоторого логического выражения для всех записей в подтаблице структурного поля и возвращает значение ИСТИНА, только если все записи в подтаблице удовлетворяют указанному условию, которое должно содержать имена полей подтаблицы.

Синтаксис:

<СтруктурноеПоле>. All | Все (<выражение>)

Для рассмотренного выше класса документов фильтр

```
"StructArray1.All(F21 = 0 and F22 = 'услуга')"
```

вернет набор записей, у которых в подтаблице StructArray1 все строки удовлетворяют заданному условию.

Количество / Count

Квантор **Количество** позволяет в выражении фильтра получить размер массива (структурного поля), то есть число записей в подтаблице конкретного документа.

Синтаксис:

<СтруктурноеПоле> . Количество | Count

Например, фильтр

```
"StructArray1.Count > 1"
```

отберет только те документы, у которых в подтаблице StructArray1 содержится больше одной строки.

Владелец / Owner

Данный квантор позволяет обращаться к полям основной таблицы (записи) из выражения, находящегося в скобках кванторов **Exists** и **All**.

Синтаксис:

Owner | Владелец . <ПолеОсновнойТаблицы>

Например, для вышеприведенного класса документов Doc1 фильтр

```
"StructArray1.All(F21*Owner.F1 > 20)"
```

предписывает отобрать те записи, для которых во всех строках подтаблицы StructArray1 выполняется следующее условие: произведение поля F21 подтаблицы и поля F1 основной записи больше 20.

Несмотря на удобство механизма кванторов, пользоваться ими нужно осмотрительно, так как массовое использование условий на поля подтаблиц может заметно замедлить фильтрацию документов.

Для редактирования шаблона (TPL-файла) картотеки используется [Визуальный редактор шаблонов бланков](#). Общие принципы редактирования шаблона картотеки практически ничем не отличаются от тех, что описаны для бланка. Основное отличие заключается в том, что контекстное меню шаблона дополнительно содержит две команды **Свойства картотеки** и **Свойства столбца**. Более подробно они рассматриваются в темах:

- [Общие свойства картотеки](#)
- [Свойства столбцов](#)

Кроме того, шаблон картотеки имеет, как правило, несколько областей, в отличие от шаблона бланка, который имеет только одну область. Чисто визуально это проявляется в том, что окно картотеки разделено вертикальными или горизонтальными перемычками на несколько частей. Перемычки можно двигать мышью, увеличивая или уменьшая любую область.

Каждая из областей имеет специфическое назначение. Главная из них содержит непосредственно таблицу с данными из картотеки. Эта область присутствует в окне всегда. Если в картотеке разрешено иерархическое представление записей и в настройках указано [показывать иерархию в виде дерева](#), то слева от табличной части выводится область с иерархией имеющихся групп записей.

Еще одна область, которая по умолчанию размещается у правого края окна картотеки, предназначена для добавления на шаблон произвольных элементов управления: кнопок, полей ввода, флагов и т.д. Положение данной области может переноситься разработчиком формы к любому краю окна картотеки. Для ее перемещения необходимо навести курсор мыши на перемычку и нажать правую кнопку. В открывшемся контекстном меню доступны команды **Слева**, **Справа**, **Снизу**, **Сверху**, **Скрыть**, которые и позволяют управлять положением и видимостью области. Первые четыре из них являются командами-переключателями – слева от каждой из них выводится условное изображение способа расположения области, причем лишь один значок отображается "нажатым", сигнализируя, какой именно режим сейчас выбран. Когда область скрыта, ее перемычка находится рядом с соответствующим краем окна, и для того, чтобы показать область вновь, достаточно переместить перемычку от края методом Drag'n'Drop.

Еще одна область предназначена для отображения содержимого подтаблиц, то есть структурных массивов значений, хранящихся внутри записей. Например, в накладной имеется табличная часть с перечнем позиций. Можно настроить картотеку таким образом, чтобы при выделении некоторой накладной (строки) в основной области окна картотеки позиции из накладной выводились в другой области, ответственной за представление подтаблицы. Разумеется, работа с подтаблицами (см. раздел [Добавление, удаление и размещение подтаблиц](#)) возможна только в том случае, если в картотеке с помощью [MTL-описания](#) определена одна или более подтаблиц, то есть массивы [многозначных полей](#) или [многозначных структур](#). [Свойства самих подтаблиц](#) определяются в соответствующем диалоговом окне, вызываемом по команде **Свойства картотеки из области подтаблицы**.

Область подтаблицы, также как и область элементов управления, может быть расположена у любого края окна картотеки или скрыта. По умолчанию она находится у нижнего края окна. Разработчик имеет возможность разместить области по-другому, используя команды контекстного меню, как описано выше.

Диалог "Свойства картотеки" позволяет настроить свойства картотеки и содержит следующие страницы: [Общие](#), [Вид](#), [Правка](#) и [События](#). Для его открытия необходимо выполнить команду [Свойства картотеки](#) контекстного меню.

Страница "Общие"

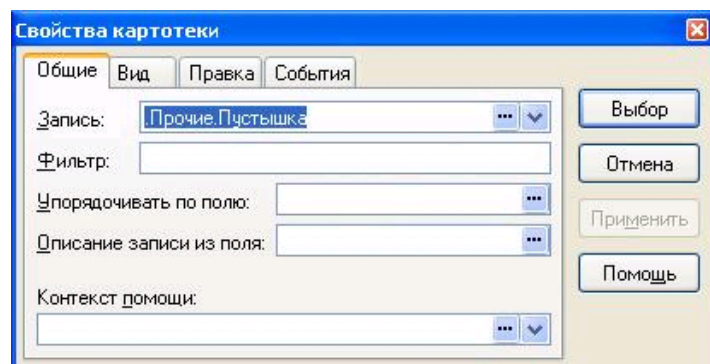


Рис. Страница "Общие" свойств картотеки.

Поле **Запись**

В это поле из выпадающего списка вводится идентификатор записи, для которой создана данная картотека. В списке можно выбрать один или более классов записей (для гетерогенной картотеки). Требуемые классы записей помечаются в списке флагами. В поле ввода имена выбранных классов перечисляются через запятую.

Если картотека гетерогенная, то есть создана для нескольких классов записей, то в выпадающем списке поля **Запись** рядом с каждым классом записи располагается флаг, с помощью которого и определяется, является ли конкретная запись источником данных для картотеки. Когда флаг включен, запись используется в картотеке.

В поле **Запись** можно ввести имя специального системного класса записей Kernel.Отчеты - именно такой класс записей используется для хранения информации об [общих пользовательских отчетах](#). После компиляции и запуска проекта в такой картотеке станет возможным добавление столбцов для полей класса записи Kernel.Отчеты.

Поле **Фильтр**

В поле можно ввести произвольное выражение логического типа, позволяющее ограничить набор отображаемых в картотеке записей лишь теми, которые удовлетворяют указанному условию.

Поле **Упорядочивать по полю**

В поле задается сортировка документов картотеке по конкретному полю (из числа имеющихся). Ввод в поле производится вручную или из списка (кнопка).

Поле **Описание записи из поля**

Данное поле предназначено для иерархических картотек, в которых включен показ дерева. Если оставить данное поле пустым, то по умолчанию в качестве имен групп в дереве берутся **DocID** соответствующих групповых записей. Если же в данном поле указать одно из полей записи, то именно его содержимое будет отображаться в дереве картотеки в качестве имен групп. Кроме того, это же поле для всех картотек служит источником обозначения записи, которое выводится пользователю при попытке выполнить над записью операцию, требующую подтверждения. В этом случае описание записи (т.е. содержимое указанного поля записи) выводится в диалоговом окне подтверждения.

Иными словами, значения указанного поля используются в качестве идентификаторов групп. Совокупность значений из нескольких последовательно вложенных групп, определяющих положение выделенного документа, выводятся в виде последовательности идентификаторов в строке состояния. Фактически, это аналог пути файла в файловой системе, то есть – путь документа в иерархии картотеки.

Поле **Контекст помощи**

В поле указывается путь к файлу с темой справки. Для ее открытия следует в окне картотеки нажать клавишу **F1**.

Страница "Вид"

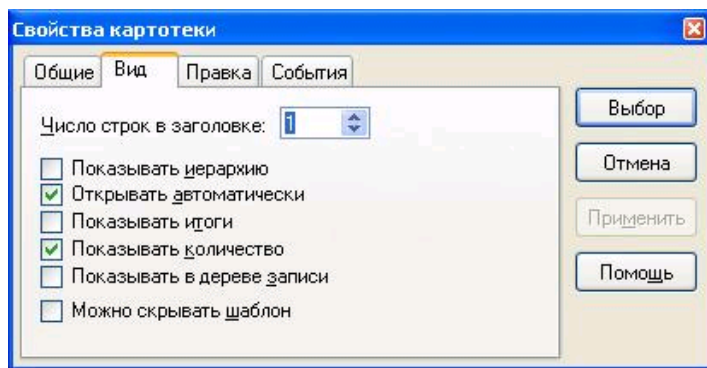


Рис. Страница "Вид" свойств картотеки.

Поле **Число строк в заголовке**

С помощью данного поля можно указать высоту шапки табличной части картотеки.

Флаг **Показывать иерархию**

Флаг определяет способ отображения записей в иерархических картотеках, которых в MTL-описании указан модификатор [Hierachical](#). Напомним, что в иерархической картотеке допустимо создавать группы документов произвольной вложенности.

Если флаг установлен, то в каждый момент времени в окне картотеки будут отображаться записи текущего уровня (одной группы или корня картотеки). Если же флаг снят, то отображаются все записи картотеки из всех групп, вне зависимости от их уровня вложенности. Это, так называемое, плоское представление иерархической картотеки.

Внимание. Если для неиерархической картотеки установлен этот флаг, то при открытии ее в режиме сессии выдается сообщение об ошибке "Неизвестный идентификатор GroupDoc".

Для включения видимости дерева в иерархических картотеках достаточно подвести курсор мыши к левому краю табличной части окна картотеки и задержать его над перемычкой, разделяющей область с деревом и таблицу (по умолчанию область с деревом свернута и перемычка находится у левого края таблицы). Над перемычкой курсор меняет свою форму на две вертикальных черты с разнонаправленными стрелками. Нажав левую кнопку мыши и не отпуская ее, разработчик может переместить перемычку вправо, задавая тем самым ширину области с деревом иерархии.

Флаг **Открывать автоматически**

Флаг используется для динамически созданных объектов картотеки. По умолчанию флаг включен, в этом случае при открытии картотеки запрос выполняется автоматически, и в окне картотеки отображаются все записи. При отключенном флаге открывается картотека, не содержащая записей, а записи появляются на экране по явному запросу. Такой режим записи удобен при больших картотеках, когда требуется отображать не все записи, а только часть записей, отфильтрованных по заданному условию, чтобы не терять время на открытие всей картотеки.

Флаги **Показывать итоги, Показывать количество**

В случае если флаги установлены, показывается итоговая строка и количество записей в картотеке.

Флаг **Показывать дерево в записи**

При установке данного флага в дереве картотеки отображаются не только группы, но и входящие в них записи, иначе - отображаются только группы.

Флаг **Можно скрывать шаблон**

По умолчанию флаг выключен для совместимости со старыми картотеками. Установка флага позволяет пользователю во время сессии интерактивно управлять с помощью мыши шириной (и видимостью) шаблонной части картотеки, и, даже сделать шаблонную часть картотеки невидимой.

Страница "Правка"

На странице "Правка" в группе **Разрешено** расположены флаги, каждый из которых разрешает или запрещает пользователю выполнить конкретное действие с картотекой.

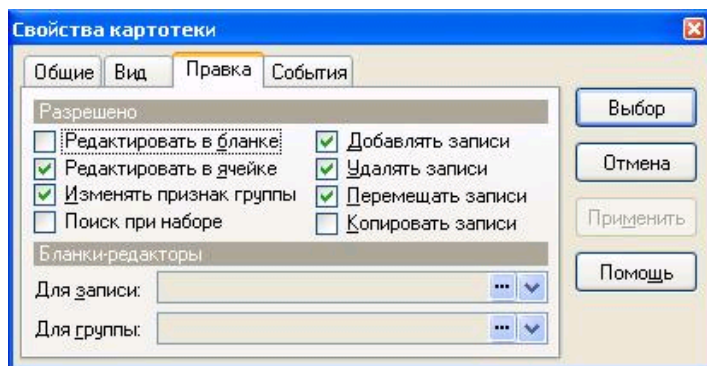


Рис. Страница "Правка" свойств картотеки.

Флаг **Редактировать в бланке**

Установка флага разрешает открывать бланк-редактор для ввода и изменения записей картотеки. Сам бланк-редактор (для одиночной записи и для группы записей) назначается с помощью полей ввода в группе **Бланки-редакторы** (см. далее).

Флаг **Редактировать в ячейке**

Флаг позволяет изменять поля записей непосредственно в ячейках таблицы картотеки. Если флаг включен, то по нажатию клавиши **Enter** или щелчку мыши в ячейке появляется так называемый inplace-редактор, то есть поле ввода, которое содержит текущее значение ячейки и позволяет его изменить. При снятом флаге пользователь не может открыть inplace-редактор и отредактировать значение в ячейке картотеки.

Замечание. Когда оба флага сняты, интерактивное редактирование запрещено, хотя доступны программные средства редактирования.

Флаг **Изменять признак в группе**

При включенном флаге пользователю разрешается изменять статус записи, т.е. групповая запись может стать простой и наоборот. При снятом флаге статус записи изменить нельзя.

Флаг **Поиск при наборе**

Если флаг установлен, то в режиме редактирования (inplace-редактор) при нажатии на любую клавишу, открывается диалог поиска в картотеке, а чтобы изменить данные в ячейке, предварительно необходимо нажать клавишу **Enter**.

Внимание. Если текущий документ (картотека) уже находится в состоянии редактирования, то флаг игнорируется.

Флаги **Добавлять записи**, **Удалять записи**, **Перемещать записи** и **Копировать записи**

Флаги позволяют включить или запретить выполнение операций с записями: добавления, удаления, перемещения и копирования. Если соответствующий флаг установлен, то операция разрешена. Под перемещением понимается перенос записи из одной группы в другую методом drag'n'drop или с помощью команд вырезания и вставки.

Группа полей **Бланки-редакторы**

С помощью группы элементов **Бланки-редакторы** можно назначить для картотеки бланки-редакторы, то есть бланки, которые будут вызываться из картотеки при попытке отредактировать какой-либо документ. Существует возможность указать два разных бланка-редактора: бланк для отдельной записи и бланк для групповой записи – они выбираются из выпадающих списков **Для записи** и **Для группы**. При необходимости оба поля могут быть заполнены или оставлены пустыми, или же может быть назначен лишь один из бланков. Например, если из выпадающего списка **Для группы** выбран конкретный класс бланков, то именно этот бланк будет использоваться для редактирования документов, которые описывают (представляют собой) группу.

В каждом из этих списков перечисляются зарегистрированные в проекте бланки, описанные как бланки-редакторы текущей записи (типа документов). Следует иметь в виду, что из списка (как для простых бланков-редакторов, так и для бланков-редакторов групп) могут быть выбраны сразу несколько бланков – для этого достаточно пометить их флажками, расположенными у левого края каждого элемента списка. Если для класса документов назначено более одного бланка редактора (имеется в виду – более одного для каждого из двух видов), то при попытке отредактировать запись в картотеке в процессе выполнения проекта пользователю будет выдано диалоговое окно с перечислением всех имеющихся бланков-редакторов, назначенных для данной картотеки. Лишь, после того как пользователь выберет в этом диалоге требуемый бланк, последний открывается на экране с загруженной для редактирования записью.

Страница "События"

Страница "События" аналогична странице [настройки событий в бланке](#), однако, очевидно, что на ней перечисляются и определяются [обработчики событий](#), специфических для картотеки.

Свойства подтаблицы

Свойства подтаблицы задаются с помощью одноименного диалога, который вызывается двойным щелчком мыши в свободной области подтаблицы (если выполнить двойной щелчок в какой-либо клетке, будет вызван диалог свойств соответствующего столбца подтаблицы). В диалоге имеется 3 страницы: "Общие", "Правка" и "События".

На странице "Общие" в поле **Структура** выводится название поля (структуры), из которого будут браться данные для подтаблицы. Выпадающий список **Упорядочивать по полю** позволяет определить, по какому полю структуры будут отсортированы строки в подтаблице. Ниже расположен выпадающий список для указания количества строк в заголовках колонок (то есть, высоты шапки).

Поле **Ширина/высота** позволяет ввести размер подтаблиц в том случае, если их в записи больше одной. Если в поле введен ноль, подтаблицы делят выделенное под них место пропорционально их количеству. Если в записи существует только одна подтаблица или речь идет о свойствах последней из подтаблиц, поле **Ширина/высота** не используется.

На странице "Правка" с помощью соответствующих флагов можно запретить или разрешить модификацию строк в подтаблице, добавление и удаление строк, выделение групп строк и копирование групп строк.

Страница "События", аналогичная странице [настройки событий в бланке](#), позволяет определить обработчики событий, специфических для подтаблиц. Перечень событий подтаблицы картотеки приведен в описании класса [ПодтаблицаКартотеки/SubCardfile](#).

Назначение диалога: задание и редактирование свойств столбцов картотеки. Данный диалог открывается командой контекстного меню **Свойства столбца** и размещается на следующих страницах: "[Общие](#)", "[Формат](#)", "[Шрифт](#)", "[Подсказка](#)", "[События](#)".

Страница "Общие"

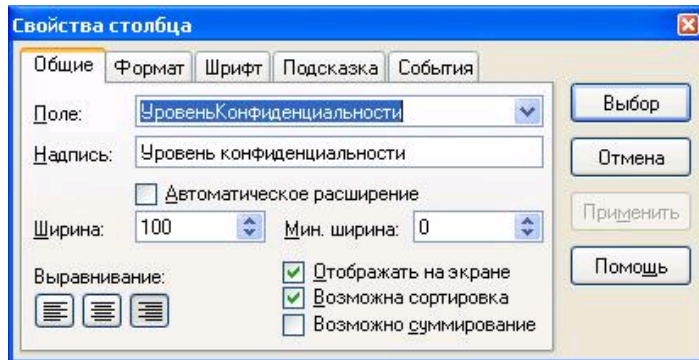


Рис. Страница "Общие" свойства столбцов.

Поле **Поле**

В поле выводится идентификатор поля записи, которое пользователь выбрал для отображения в текущем столбце. Для вычисляемых столбцов – это произвольный идентификатор (отличный от идентификаторов остальных полей). Выпадающий список, связанный с полем, позволяет быстро перейти к редактированию свойства другого столбца – достаточно выбрать его из списка. Изменения, сделанные в текущем столбце, автоматически сохраняются перед переключением на другой столбец.

Поле **Надпись**

Заголовок столбца, видимый пользователю на стадии исполнения проекта, может быть произвольным и задается в этом поле. Если текст заголовка слишком длинный, чтобы уместиться в столбце по ширине, программа может автоматически переносить его по словам на следующие строки, а также в явном виде задать перевод строки в заголовке столбца. Это достигается вставкой символа ";" (точка с запятой) в требуемое место заголовка. В результате сам символ ";" отображаться не будет, но в данном месте заголовка произойдет перевод строки. Следует иметь в виду, что в заголовке может встречаться символ переноса (переносом считается тире после алфавитно-цифрового символа и перед последовательностью, состоящей из пробела либо точкой с запятой и опять же алфавитно-цифрового символа, например, "перенос" или "пере-;нос"). Символ переноса является чисто визуальным элементом – он отображается на экране, но не вызывает переноса строки заголовка, однако в меню и списках, где заголовок должен всегда отображаться одной строкой (слитно), программа автоматически будет убирать символ переноса.

Флаг **Автоматическое расширение**

Флаг позволяет управлять режимом автоматического подстраивания ширины столбца. Когда флаг установлен, ширина столбца определяется самой программой, исходя из размеров окна картотеки и ширины остальных столбцов. Общая ширина всех столбцов с автоподстраиваемой шириной вычисляется программой как разница между шириной окна картотеки и суммой ширин остальных столбцов (для которых задан конкретный размер). Затем из общей ширины автоподстраиваемых столбцов простым делением на их общее число определяется ширина одного столбца. Например, если все столбцы в картотеке определены подобным образом, то ширина каждого из них будет равна общей ширине окна, деленной на число столбцов. Если ширина окна картотеки недостаточна для того, чтобы вместить все столбцы, автоподстраиваемые столбцы вырождаются в узкую вертикальную полоску, если в поле **Мин. ширина** не задана минимальная ширина.

При этом также существует возможность включать/отключать автоматическую подстройку ширины любого столбца с тем, чтобы общая ширина всех столбцов была подогнана под ширину окна картотеки. Для этих целей необходимо подвести курсор мыши к правой границе шапки столбца и сделать двойной щелчок мыши. Последующая смена ширины столбца (методом "перетаски и отпусти") отключает автоподбор ширины. В картотеке может быть несколько столбцов с автоподбором ширины.

Поле **Ширина**

Поле предназначено для задания ширины столбца в пикселях. Когда флаг **Автоматическое расширение** включен, данное поле становится недоступным, а указанное в нем значение игнорируется. Кроме того, пользователь может изменить ширину любого столбца на стадии выполнения проекта методом "перетаски и отпусти". Для этого достаточно подвести курсор к границе столбца в шапке картотеки, нажать кнопку мыши и, не отпуская ее, перемещать мышью в требуемую сторону, что приведет к расширению или сужению столбца.

Поле **Мин.ширина**

В этом поле можно ввести минимальную ширину столбца. При автоматическом подборе ширины (флаг установлен) программа не будет сужать столбец сильнее, чем указано здесь. Ограничение также действует, если пользователь сам изменяет размер столбца.

Группа кнопок **Выравнивание**

В зависимости от выбранной кнопки текст клеток в данном столбце будет сдвигаться влево, вправо или располагаться симметрично относительно центра клетки.

Флаг **Отображать на экране**

Если флаг включен, то столбец по умолчанию будет отображаться в окне картотеки, в противном случае – будет невидим. Управлять видимостью столбцов можно не только на стадии проектирования, но и в режиме исполнения проекта, если в [схеме доступа](#) конкретному пользователю разрешено настраивать картотеку.

Флаг **Возможна сортировка**

Флаг позволяет определить, можно ли будет отсортировать информацию по текущему столбцу. Если флаг включен, сортировка разрешена (она выполняется по нажатию кнопки мыши на заголовке столбца).

Флаг **Возможно суммирование**

Флаг необходим для вывода на экран и на печать специальной строки с подсчетом итогов по столбцам. Данный флаг рекомендуется установить у тех столбцов, которые содержат числовые величины. Установка флага означает лишь возможность суммирования, а непосредственно суммирование выполняется после вызова команды [Картотеки|ПоказыватьСуммы](#). Суммы выводятся в специальной строке у нижнего края таблицы, причем по щелчку мыши в крайней левой (серой) клетке этой строки происходит последовательное переключение между подсчетом суммы и нахождением минимального или максимального значения в колонке. Для колонок со значениями типа **Дата** подсчет сумм невозможен, однако минимальное и максимальное значения находятся, поэтому для таких колонок также имеет смысл включать флаг **Возможно суммирование**.

Страница "Формат"

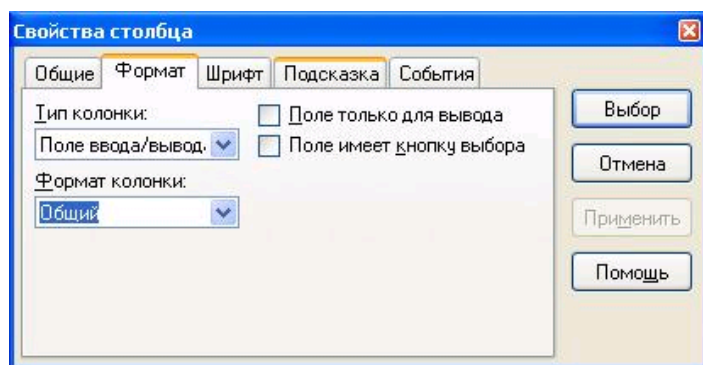



Рис. Страница "Формат" свойства столбцов.

На странице "Формат" можно установить тип и формат колонки, а также задать, используется ли данный столбец для ввода и/или вывода данных.

Флаг **Поле только для вывода**

При установке флага редактировать значение в клетках столбца запрещено, ввод осуществляется программным способом.

Флаг **Поле имеет кнопку выбора**

Когда включен флаг, у правого края текущей клетки столбца будет появляться кнопка , по нажатию которой будет формироваться событие **ПриОбзоре** и может выполняться стандартное действие (которое зависит от типа поля), если это не было запрещено в обработчике события.

Поле **Тип колонки**

Поле предназначено для выбора типа текущей колонки из предлагаемых вариантов:

- **Поле ввода/вывода** - поле, тип которого задается в поле **Формат колонки**;
- **Вычисляемое поле** - поле, ввод и вывод в которое осуществляются на прикладном уровне с помощью обработчиков событий, поле не связано ни с одной переменной.

Поле **Формат колонки**

Ввод в поле производится из выпадающего списка, в котором перечислены разрешенные в языке варианты форматов полей колонок. Для каждого из имеющихся форматов указываются особенности их

использования:

- **Общий** - поле ввода-вывода без конкретизации типа. Программа автоматически присвоит полю тот тип, который имеет переменная в описании бланка;
- **Строка** - поле строкового типа. При выборе этого формата отображается поле **Максимальное число символов**, в котором можно задать *ограничение на число символов, вводимых при редактировании полей картотеки*;
- **Число** - поле числа, используемое для целых и числовых переменных. Для числовых полей дополнительно появляется поле **Формат вывода числа**, в выпадающем списке которого перечислены наиболее употребительных форматы, которые удобно использовать для получения унифицированного способа отображения значений в столбце. Для выбранного формата отображается внешний вид представления значений в столбце. Стандартный формат числа предусматривает использование точки (".") в качестве разделителя целой и дробной части. Кроме того, можно выбрать формат числа, в котором в качестве разделителя троек разрядов (триад) числа будет использоваться апостроф, например, 3'456.23. Допустимые символы форматного преобразования и их назначение рассматриваются в теме ["Преобразования формата"](#).
- **Дата** - формат полей для задания дат. В этом случае также нужно выбрать формат их вывода в поле **Формат вывода даты** и варианты вывода даты (Только дата, Только или дата и время) в поле **Ввод даты**;
- **Логический** - поле логического типа, в этом случае в клетках столбца будет выводиться значок (флаг), причем в поле **Формат отображения** можно указать способ отображения значений (Показывать оба значения - задано по умолчанию, Только Истину, Только Ложь);
- **Перечислимый** - поле перечислимого типа, которое может быть связано с переменной целого или логического типа данных и списком строк, обозначающих альтернативные значения, которые указываются в поле **Варианты строк**;
- **Ссылочный** - поле ссылочного типа, используемое для отображения ссылки на другой документ или значения некоторого поля из другого документа. Значения данного ссылочного поля можно редактировать вручную, если установлен флаг **Разрешить ручной ввод**. В этом режиме в клетке столбца будет открываться inplace-редактор по нажатию клавиши **Enter** или щелчку мыши, позволяя пользователю вводить значения вручную. Ввод нового значения будет генерировать событие **ПриНаборе**. Если флаг снят, изменение значения в клетке столбца может происходить как результат стандартной обработки действий пользователя. Например, ссылочное поле обычно заполняется из картотеки документов того класса, для которого данное поле предназначено (при соответствующих настройках картотека вызывается автоматически при попытке отредактировать значение поля). Назначение полей **Картотека / Поле / Фильтр** аналогично назначению одноименных полей на странице "Формат" диалога ["Свойства клетки"](#).

Страница "Подсказка"

Поле **Текст подсказки**

В поле вводится произвольный текст, поясняющий назначение текущего столбца картотеки. Если установить курсор на заголовке столбца, то текст подсказки будет отображаться в специальном всплывающем окне.

Поле **Контекст помощи**

Если требуется более подробная информация, то нужно заполнить это поле, указав путь к файлу с темой помощи. В этом случае, если нажать клавишу **F1** и установить фокус на текущем столбце, откроется Справочная система, в окне которой отобразится заданная тема.

Страницы "Шрифт" и "События"

Назначение страниц "Шрифт" и "События" полностью аналогично назначению страниц в диалоге ["Свойства клеток"](#). Перечень событий столбца картотеки приведен в описании класса [СтолбецКартотеки](#).

В выражениях фильтров можно использовать не только поля, описанные в классе документа с помощью языка MTL, но и служебные поля, добавляемые в запись и заполняемые самой системой. Часть служебных полей является обязательной: такие поля всегда присутствуют в записях любого класса. Следует понимать, что в выражении фильтра используются именно имена *полей записи* (записи БД), а не *свойства объекта* класса [Запись / Record](#) (или прикладного производного класса).

Ниже приведена таблица с именами служебных полей, их типами и описанием. Поля, помеченные звездочкой, появляются в записи только при явном их описании в MTL-файле.

Название	Назначение	Тип
DocID	идентификатор документа в контексте таблицы БД	целое, если иное не задано явно в MTL-файле
ExtID	идентификатор документа в глобальном контексте	строка
CreateDate*	дата создания записи	дата
ModifyDate	дата последнего изменения записи пользователем или самой системой	дата
UpdateDate*	дата последнего изменения записи пользователем	дата
Deleted	признак того, что запись удалена: 0 – неудаленная запись; положительное целое – удаленная запись	целое, если иное не задано явно в MTL-файле
IsGroup	признак того, является ли запись группой: -1 – это группа; 0 – это простая запись	целое, если иное не задано явно в MTL-файле
GroupDoc	DocID документа-группы	должен совпадать с типом DocID
CreateUser*	имя пользователя, создавшего документ	строка
UpdateUser*	имя пользователя, последним изменившего документ	строка

Важно отметить, что ключевое поле, которое содержит внутренний идентификатор документа и по умолчанию имеет название **DocID**, может по [явному указанию разработчика](#) иметь другое имя. При этом, однако, данное поле по-прежнему будет доступно в [фильтрах](#) (в том числе [запросов](#) и [картотек](#)) только под псевдонимом **DocID**.

В фильтрах картотек допускается использовать следующие функции встроенных классов **Система** и **Бухгалтерия**:

- [Div](#)
- [Mod](#)
- [Бол / Up](#)
- [Время / Time](#) *
- [Год / Year](#)
- [Деб / Deb](#)
- [День / Day](#)
- [Длина / Length](#)
- [Корень / Sqrt](#)
- [Кре / Cre](#)
- [Лог / Log](#)
- [Макс / Max](#)
- [Мал / Lo](#)
- [Мес / Mon](#)
- [Мин / Min](#)
- [Минута / Minute](#)
- [Окр / Round](#)
- [Отбр / Trunc](#)
- [ПериодСек / PeriodSec](#)
- [Повтор / RepStr](#)
- [ПодСтр / SubStr](#)
- [Поз / Pos](#)
- [Сегодня / Today](#) *
- [Сейчас / Now](#) *
- [Секунда / Second](#)
- [Соотв / Match](#)
- [Стр / Str](#)
- [Цел / Int](#)
- [Час / Hour](#)
- [Числ / Num](#)
- [Эксп / Exp](#)

Три функции, связанные со временем и помеченные символом *, вычисляются в фильтрах лишь один раз – в начале построения запроса по записям картотеки. Таким образом, значения **Сегодня**, **Время** и **Сейчас** равны для всех просматриваемых записей, даже если выборка данных занимает заметное время.

Кроме этих функций и стандартных арифметических и логических операций в выражении фильтра можно использовать оператор **in**, проверяющий наличие указанного значения в указанном массиве. Это оператор, фактически, эквивалентен функции **SearchInArray**, не доступной в фильтрах напрямую.

Для ссылочных полей (и выражений ссылочного типа) в фильтрах имеется возможность получить имя класса объекта, хранящегося в этом поле. В частности, поле записи с "мягкой" ссылкой (т.е. поле, описанное в MTL как *inherited record* или *inherited <class>*) может хранить объект любого производного или базового класса записи, а в фильтре может быть полезным отбор по конкретному классу записи. Для этих целей предназначен метод (функция базового класса **Объект**) **ClassType** / **ТипКласса**, вызываемый в применении к объекту:

<выражение ссылочного типа>.ТипКласса

Например, в простейшем случае проверка на соответствие имени класса объектов в поле F1 отфильтровываемых записей выглядит так:

F1.ClassType = Документы.Накладные.ТоварноТранспортная;

Здесь в поле F1, описанном в MTL как *field F1 :inherited Накладные*, может, в принципе, храниться ссылка на накладную любого типа, однако, с помощью указанного фильтра легко отобрать лишь документы со ссылками на товарно-транспортные накладные.

Бланк может быть написан особым образом с тем, чтобы иметь возможность редактировать содержимое какой-либо картотеки. Такой бланк называется *бланком-редактором картотеки*.

Признаком бланка-редактора является наличие в его [заголовке](#) атрибута **Редактор**, после которого задано [имя картотеки](#). Таким образом, бланк может быть назначен редактором лишь для одной заранее определенной картотеки. В то же время картотека может иметь несколько бланков-редакторов.

Указание атрибута **Редактор** в заголовке бланка означает, что он получает доступ помимо собственных переменных (описанных явно) к полям картотеки. Причем происходит это в неявном виде. Например, если в записи картотеки есть поле "Фамилия", то в коде бланка-редактора данной записи можно (без всяких деклараций) обращаться к содержимому этого поля аналогично обращению к переменной – просто использовать идентификатор Фамилия в требуемом выражении. По понятным соображениям в бланках-редакторах запрещено описание переменных, чьи имена совпадают с идентификаторами полей картотеки.

Когда бланк-редактор открыт, он настраивается на какую-либо запись картотеки, получая все необходимые значения из полей записи. Как правило, редактор вызывается из окна картотеки; в этом случае он работает с текущей записью данного окна. Если бланк-редактор открывается из списка бланков или в картотеке была выполнена команда создать новую запись, то бланк содержит новую пустую запись (фактически, ее еще нет в картотеке).

В каждый момент времени бланк-редактор работает только с одной записью картотеки.

Работа с полями картотеки

В бланке-редакторе нет никакой разницы между его собственными переменными и полями картотеки: для них также допустимы редактирование в полях шаблонов, использование в качестве операндов в процедурной части. Единственным различием является то, что при выходе из бланка или при переходе на другую запись пользователь может подтвердить или отказаться от изменений, сделанных им в картотеке, в то время как изменения в переменных бланков происходят в момент выполнения изменяющей операции.

Многозначные поля картотеки в бланке-редакторе можно рассматривать как [переменные-массивы](#). В частности, допустимо обращение к элементам подтаблицы с указанием имени [структурного поля](#), номера элемента подтаблицы и имени поля в подтаблице. Для такого обращения используется следующий синтаксис:

Пример обращения к значению поля **Номер** второй записи подтаблицы (многозначного поля) **Позиции**:

Позиции[2].Номер

Для [периодических полей](#) так же, как и для структурных, допустимо использование индексации, но индекс может иметь другие типы (например "Дата", а не только "Целое"), например,

Курс[01.01.98]

Если индекс не указан, он считается равным текущей системной дате.

Для ссылочных полей допустимо обращение к записи, на которую данное поле ссылается, даже если она находится в другой картотеке. Причем у этой записи также могут быть ссылочные поля, информацию из которых, в свою очередь, можно получить.

Для обращения по ссылкам используется следующий синтаксис:

Поставщик.Банк.КоррСчет

В данном случае поле **Поставщик** записи, на которую настроен бланк-редактор, содержит ссылку на запись в картотеке поставщиков. В свою очередь, в этой записи вместо банковских реквизитов поставщика хранится ссылка на его банк в картотеке банков. Записав подобную конструкцию, программист получает значение из переменной **КоррСчет** справочника банков, содержащее информацию о корреспондентском счете банка, обслуживающего данного поставщика.

Если ссылка не указывает ни на одну запись, то результатом такой операции в бланке-редакторе будет пустое значение соответствующего типа.

Если не уточнять ссылочное поле дополнительными точками и идентификаторами, то оно может использоваться как переменная типа [Запись](#). Ссылочное поле может быть передано в процедуру или функцию в качестве параметра, участвовать в операциях сравнения, ее значение может быть присвоено локальной переменной процедуры или функции бланка.

Поскольку значение ссылки является в достаточной степени внутренней информацией картотек и способно устаревать (например, запись, на которую указывала ссылка, была удалена), в бланках следует с особой осторожностью оперировать глобальными переменными типа **Запись** (заранее инициализировать их корректным значением в коде программы, а не использовать сохранившееся с прошлой сессии значение).

Отображение информации из картотеки в шаблоне

Для многозначных полей допустимо их представление в бланке-редакторе в виде [повторяющейся секции](#). В качестве полей секции в этом случае указываются поля подтаблицы, а сама секция в дизайнера форм связывается с соответствующим многозначным полем записи. Описанная таким образом секция будет содержать столько кадров, сколько записей находится в подтаблице, отображая каждую из них в отдельном кадре.

Поле типа "Ссылка" отображается в шаблоне бланка строкой вида "{число}", если оно содержит ссылку на какую-либо запись или "{0}", если данная ссылка не определена. Также для ссылочных полей разрешается задавать в качестве имени поля квалифицированный идентификатор, содержащий "путь" к полю в другой картотеке, как это было показано выше. Тогда в поле шаблона будет выводиться не значение указанного выше вида, а значение, взятое по данной ссылке или пустое значение, если ссылка не определена.

Таким образом, механизм бланков-редакторов позволяет пользователю представлять и редактировать данные из картотеки в развернутом виде, а программисту — эффективно управлять этим процессом.

Картотеки как аналитические справочники

Картотеки способны выступать в роли *аналитических справочников*, т.е. хранить в себе, например, списки поставщиков, которые можно использовать для ведения [аналитического учета](#) при записи проводок в журнале операций.

Информацию из картотек можно интерпретировать как содержимое [аналитического справочника](#), а каждую запись — как [аналитический признак](#).

Для определения картотечного источника признаков необходимо открыть окно редактора проекта, [создать](#) текстовый файл с расширением *.lis и отредактировать его в соответствии с правилами описания [аналитических справочников](#). В частности в описании можно задать имя поля, значение которого будет интерпретироваться как идентификатор признака, имена полей для комментария, единицы измерения и точности, а также фильтр, по которому из всей картотеки можно отобрать только нужные для аналитического справочника записи.

Картотеки, в которых хранятся списки первичных документов, можно рассматривать как журналы операций особого вида — так называемые *журналы-картотеки*. В них с каждой записью, входящей в картотеку, связывается набор [типовых операций](#), параметрами которых выступают поля картотеки. Возможно также задание типовых операций для каждой записи всех подтаблиц, входящих в картотеку. С точки зрения разработчика журнал-картотека - это механизм, обеспечивающий связь между документами и так машиной проводок (внутренним вычислителем программы).

Основным условием взаимодействия картотеки с типовыми операциями в рамках журнала-картотеки является требование соответствия имен и типов параметров операции и полей картотеки, т.е. необходимо, чтобы для каждого параметра, заданного в заголовке типовой операции, в картотеке существовало аналогичное поле.

Для того чтобы программа стала воспринимать картотеку первичных документов как журнал операций особого вида в структуре учета необходимо описать журнал-картотеку. Для создания журнала-картотеки необходимо открыть окно редактора проекта, в левой части этого окна в иерархии объектов проекта выделить ветвь Структура учета. Далее выполнить команду **Добавить** контекстного меню и с помощью диалога ["Создание структуры учета"](#) создать текстовый файл с расширением *.lis. После создания файла в нем нужно [описать карточный журнал](#) в соответствии с синтаксисом языка структуры учета,

Суть этого описания заключается в том, чтобы связать каждую запись картотеки с той или иной совокупностью типовых операций. В результате картотека документов будет обрабатываться программой как журнал, в котором в свернутом виде записаны одинаковые типовые операции, а на место параметров подставляются значения из полей записи (документа). Закончив описание, *проект нужно откомпилировать*.

Описание картотечного журнала содержит алгоритм, в соответствии с которым операционный документ генерирует набор проводок. Кроме этого, наличие интерфейсного элемента - картотеки, связанной с журналом, - позволяет визуализировать и редактировать документы (но не проводки или типовые операции).

Алгоритм формирования проводок задается в свойствах картотечного журнала в редакторе проекта в виде перечня нескольких (или одной) типовых операций, с передачей им в качестве параметров полей операционного документа. Причем, часть типовых операций может быть связана с подтаблицей (многозначной структурной секцией) документа - при обработке документа такие операции вызываются несколько раз - по разу для каждой строки подтаблицы.

Возможен вариант, когда те или иные проводки нужно сформировать не только или не столько для самих записей картотеки, а для их структурных полей, т.е. входящих в них подтаблиц. Например, необходимо сформировать определенные бухгалтерские записи для каждой позиции накладной, и при этом таблица позиций накладной хранится в структурном поле с именем "СоставНакл". Этот случай также контролируется с помощью редактора проектов, что позволяет сформировать нужные проводки по всем товарам, отпущенным по накладным, которые зарегистрированы в картотеке.

Редактирование картотек

Картотеки (в том числе картотечные журналы, и картотеки аналитических признаков) формируются из записей (документов), которые, как правило, представляют собой экземпляры заполненных бланков. Поэтому вопрос о взаимодействии картотек с бланками является наиболее важным и будет рассмотрен максимально подробно.

Программные средства предоставляют два базовых способа редактирования картотек – с помощью [бланков-редакторов](#) картотеки и непосредственно в окне картотеки. Кроме того, напомним, что встроенный язык ТБ.Скрипт и набор системных классов позволяют полностью управлять картотеками на программном уровне.

Оба этих способа описываются в соответствующих разделах:

- [Использование бланка для редактирования картотеки](#)
- [Управление картотеками из процедур и функций](#)

Содержание темы:

[Команды для работы с картотеками в проекте](#)
[Этапы создания картотек в проекте](#)
[Создание гетерогенных картотек](#)
[Создание специальных картотек](#)

Команды для работы с картотеками в проекте

Картотеки, как и другие подсистемы, создаются и регистрируются в прикладном проекте с помощью целого набора интерфейсных средств. Однако, в отличие от других подсистем, картотеки требуют более сложной процедуры их создания и подключения.

Большинство [команд](#), которые потребуется выполнить для того, чтобы начать работу с картотекой, доступны из контекстного меню в [окне редактора проекта](#). Для его открытия следует установить курсор на объекте "Формы картотек" в иерархии объектов окна редактора проекта и нажать правую кнопку мыши. Помимо команды **Добавить** (см. ниже) в меню доступны команды **Удалить** (*Del*) и **Переименовать** (*Enter*), а также команда **Добавить папку** (*Alt+Ins*). С помощью них можно удалить картотеку из проекта, изменить ее название или создать папку для хранения группы картотек.

Кроме того, непосредственное отношение к картотекам имеет часть команд, доступных из окна [администрирования](#) и связанных с низкоуровневыми сущностями, такими как база данных и информационная база. Эти команды, в принципе, играют первостепенную роль, так как позволяют подготовить фундамент для всех картотек проекта, однако в силу ярко выраженной системной специфики данных команд, они рассматриваются в главе, посвященной администрированию.

Этапы создания картотек в проекте

1. Создание MTL-файлов.

Действительно, исходя из логики функционирования картотеки понятно, что ее регистрации в системе должен предшествовать как минимум один очень важный этап – описание модели данных, то есть структуры записи, которая будет отображаться в картотеке. Описание типа записи (класса записи) выполняется на [специальном языке](#) и сохраняется в текстовом файле с расширением MTL, называемом также MTL-файлом.

Для создания MTL-файла необходимо в [окне редактора проектов](#) установить курсор в ветви **Записи** иерархии объектов проекта выполнить команду **Добавить** контекстного меню. Далее в открывшемся [диалоге](#) указать имя класса записи (типа документа).

2. Компиляция проекта.

Для создания картотек необходимо, чтобы проект и, соответственно, входящие в него MTL-описания были откомпилированы. Для этого используются команды [Проект | Компилировать](#) (*Ctrl+F9*) или [Проект | Компилировать все](#) (*Alt+F9*). Лишь после компиляции описанные в MTL-файлах записи становятся доступными в текущем проекте и появляются в [иерархии объектов](#) проекта в ветви "Классы записей".

3. Создание картотек.

Теперь прикладной программист может приступить непосредственно к *созданию картотеки на основе имеющейся записи*. Все шаги по созданию картотеки выполняются в многостраничном диалоге под руководством [Мастера](#), который запускается командой **Добавить** (*Ins*) из контекстного меню.

Замечание. На основе одной и той же записи может быть создано несколько картотек, что дает возможность взглянуть на данные под разными углами зрения.

Создание гетерогенных картотек

Допускается настроить картотеку для отображения/редактирования сразу нескольких классов записей. Такие картотеки называются *гетерогенными*. В них разрешается создавать столбцы для работы с одноименными полями записей, причем одноименные поля должны иметь один и тот же тип. Например, если в проекте есть записи РасходнаяНакладная и ПриходнаяНакладная с одинаковыми полями Сумма, Номер и Контрагент, то на их основе можно создать картотеку, содержащую столбцы для этих полей.

Для создания гетерогенной картотеки необходимо на второй странице [Мастера](#) заполнять поле **Доступные записи** несколько раз и после каждой записи ставить "," (запятую). По умолчанию, в картотеке создаются столбцы для одноименных полей выбранных записей, причем лишь тех полей, которые описаны на верхнем уровне иерархии документов (поля в подтаблицах не анализируются). Впоследствии при настройке свойств картотеки разработчик может добавить столбцы для полей вложенных структур документов, а также для свойств полей ссылочного типа. Например, если в классах используемых записей описано поле **Сделка** типа **ДокументДоговора**, а в этом типе в свою очередь фигурирует свойство **Контрагент**, то в картотеку можно будет добавить столбец для поля **Сделка.Контрагент**. Если в выбранных классах записей нет явным образом описанных одноименных полей, то в картотеку вставляется единственная колонка для поля **DocID**, которое

всегда присутствует во всех классах записей.

Создание специальных картотек

Следует иметь в виду, что система позволяет создавать специальные картотеки, предназначенные для работы с пользовательскими общими отчетами (добавляемыми пользователями в диалоге "Отчеты" в режиме сессии). Подобная картотека дает возможность получить доступ к списку общих пользовательских отчетов из кода. Её создание требует следующих действий. Следует создать картотеку на базе любого из существующих классов записей, затем удалить из неё все столбцы и в [диалоге свойств картотеки](#) ввести в поле **Записи** имя специального системного класса записей: Kernel.Отчеты – именно такой класс записей используется для хранения информации об общих пользовательских отчетах. После компиляции и запуска проекта в этой картотеке станет возможным добавление столбцов для полей класса записи Kernel.Отчеты.

При решении различных расчетных задач часто требуется получить доступ к данным картотеки не для визуализации, а для использования в расчетах. В таких случаях использование бланков-редакторов было бы слишком громоздким и неудобным. Поэтому язык ТБ.Скрипт включает в себя широкий набор процедур и функций для доступа к картотечным данным, реализованный в виде нескольких классов ([Картотека / CardFile](#), [Запись / Record](#), [ФормаКартотеки / CardForm](#), [Структура / Structure](#), [Подтаблица / Subtable](#) и др.).

С помощью данных классов картотеку можно упорядочить по тому или иному признаку, наложить на нее *фильтр*, т.е. отобразить только записи, удовлетворяющие определенному условию, провести поиск нужной информации.

Следует четко разграничивать понятия **картотеки** и объектов класса **Картотека** (или **ФормаКартотеки**). Картотека существует в базе данных в единственном экземпляре. Объектов же может быть создано несколько, причем все они могут существовать одновременно. Важно, что за счет наличия таких дополнительных параметров как фильтр, сортировка по определенному полю и т.д. информация из одной и той же картотеки может быть представлена в разных объектах по-разному. Другими словами, картотечные объекты реализуют своего рода "окна", через которые можно "смотреть" на данные картотеки под разными углами.

Отчеты позволяют анализировать информацию о финансово-хозяйственной деятельности предприятия в различных разрезах и могут создаваться как программистами в режиме проектирования (так называемые *проектные отчеты*), так и рядовыми пользователями в режиме сессии. Проектные отчеты, разработанные в редакторе проекта, могут использоваться в качестве базовых для пользовательских отчетов, созданных в режиме сессии.

Все отчеты как проектные, так и пользовательские отображаются в диалоге ["Внутренние отчеты"](#) в иерархическом виде и могут образовывать дерево неограниченной глубины. В дереве отчетов пользовательские отчеты визуально отличаются только видом иконки (наличием изображения руки для общих отчетов и лица - для локальных отчетов).

В проекте внутренние отчеты регистрируются на равных правах с остальными составляющими проекта. Они имеют пользовательский и программный интерфейс, позволяющий управлять форматом выходных данных и логикой работы отчетов как на стадии проектирования (через диалоговое окно [настройки свойств](#) похожее на диалог "Внутренние отчеты"), так и во время исполнения проекта (через обращения к свойствам класса [Отчет](#)). Свойства и методы этого класса обеспечивают доступ к настройкам и результирующей информации отчета из программы на ТБ.Скрипт, что позволяет производить избирательную выборку данных и их постобработку перед выводом на экран.

Дополнительно существует возможность визуализации отчетных данных с помощью классов [ФормаОтчета](#) и [ГрафикОтчета](#). С помощью интерфейсных средств программы (диалоги настройки свойств, команды групп "Учет" и "Графика") функционал этих классов доступен не только программистам, но и конечным пользователям, строящим отчеты непосредственно в режиме сессии (если это разрешено схемой доступа). Так, создание отчетов и настройка их общих свойств выполняется в диалоге, вызываемом командой **Учет|Отчеты** (этот диалог аналогичен тому, что используется программистами на стадии проектирования), а настройка деловой графики производится с помощью диалога ["Настройки графического отчета"](#).

Пример использования переменной типа **Отчет** для доступа к информации внутреннего отчета из прикладного класса ТБ.Скрипт приведен в темах:

[Пример работы с объектом Отчет;](#)

[Пример работы с иерархическим отчетом.](#)

Сведения, необходимые для настройки и работы с внутренними отчетами описаны в подразделе ["Отчеты"](#) в разделе "Инструменты". Вопросы разработки отчетов на шаблонах описаны в данном подразделе в темах:

- [Разработка шаблона для отчета](#)
- [Шаблон для отчета по оборотам](#)
- [Шаблон для отчета по проводкам](#)
- [Формирование клеток секций отчета на шаблоне](#)
- [Особенности построения отчетов на шаблоне с фреймами](#)

Особенности построения отчетов на шаблоне с фреймами

Для построения отчетов разрешается использовать как шаблоны, имеющие только один корневой фрейм с предопределенным именем **RootFrame**, так и шаблоны, состоящие из нескольких фреймов. Предопределенные секции шаблона отчетов можно распределять по различным фреймам, при этом секцию "ЗаголовокТаблицы" и саму таблицу желательно располагать в одном фрейме.

Отчет на шаблонах строится по умолчанию в том случае, когда шаблон для него считается пустым, а именно, если есть только один корневой фрейм и в нем нет ни одной секции. При наличии нескольких фреймов, отчет будет строиться в том фрейме, где расположена заданная предопределенная секция для вывода. Однако, если предопределенная секция для вывода не задана на шаблоне, то необходимо в явном виде задать предопределенное имя фрейма **ФреймОтчета** или **ReportFrame** в поле **Имя** на странице "Общие" диалога ["Свойства фрейма"](#).

В тех случаях, когда в шаблоне, кроме корневого, имеются другие фреймы и не найдена предопределенная секция для вывода отчета, а также не задано предопределенное имя фрейма **ФреймОтчета|ReportFrame**, то возбуждается исключение "Не указан фрейм для вывода отчета", т.к. программе не известно, в каком фрейме строить отчет. Причем, если указан фрейм с именем ФреймОтчета|ReportFrame и в нем нет ни одной секции, то исключение не возбуждается.

Класс отчета состоит из 3-х файлов: rpt-файла с описанием настроек отчета, а также cod-файла и tpl-файла, обеспечивающих, при необходимости, алгоритмическую часть и интерактивный интерфейс для пользователя. Вообще говоря, и cod-файл, и шаблон могут быть пустыми (или вообще отсутствовать). Они однозначно не требуются, если отчет строится в форматах, отличных от формата шаблона и график. Однако и в этих двух случаях, шаблон и код не обязательны - при их отсутствии система динамически конструирует стандартный шаблон в момент построения отчета и выводит в него результаты.

Следует отметить, что для графического отчета шаблон используется исключительно для обеспечения интерактивного взаимодействия с пользователем (например, для размещения дополнительных элементов управления, изменяющих по запросу пользователя внешнее представление данных на графике), но не для вывода результатов отчета. Если отчет строится *в формате шаблона*, то его шаблон содержит таблицу с результатами отчета и может редактироваться разработчиком в целях изменения структуры и стиля отображения информации.

При построении отчета на шаблоне система использует следующую логику поиска шаблона. Если вместе с rpt-файлом лежит и одноименный tpl-файл, то берется он. Если такого шаблона нет, система ищет в каталоге Bin файл report.tpl, и если находит, то применяет его. Если же и такого общего шаблона не найдено, то система динамически генерирует шаблон со структурой по умолчанию.

Фактически разработка шаблона для отчета сводится к размещению на нем секций с предопределенными именами, известными системе (см. ниже). Не обязательно все из этих секций должны быть на шаблоне.

Заполнение шаблона результатами отчета происходит по принципу поиска поименованных секций и столбцов, имена которых зарезервированы, а структура должна отвечать определенным стандартам. Клетки найденных секций заполняются показателями, которые выбираются либо по явному указанию разработчика (с помощью специальных макросов), либо исходя из соответствия внутренней структуры данных отчета и секции.

Если какой-либо из зарезервированных секций нет, то соответствующая порция данных просто не отображается в отчете. Если же кроме предопределенных секций на шаблон добавлены секции с другими произвольными именами, то все они будут выведены в верхней части шаблона (перед секциями с результатами отчета), а заполнение таких секций лежит полностью на алгоритмической части класса отчета (т.е. оно не выполняется автоматически). Дополнительно на шаблоне можно разместить и любые элементы управления.

Перечень секций, используемых в шаблоне отчета

В отчетах на шаблонах могут использоваться перечисленные в таблице секции с предопределенными именами (в скобках приведены их английские синонимы), автоматически заполняемых системой. Для каждой секции указан требуемый размер и вид отчета, в котором она используется.

Секция	Назначение	Вид отчета	Строк	Столбцов
Название (Title)	название отчета	любой	1	1
Период (Period)	даты отчетного периода	любой	1	1
План (Plan)	план счетов	любой	1	1
Счета (Accounts)	условие отбора счетов	любой	1	1
Параметры (Parameters)	условие отбора признаков	любой	1	1
УточняемыеПараметры (PrecParams)	уточняемая аналитика	уточняющий	1	1
ЗаголовокТаблицы (TableHeader)	заголовок таблицы по оборотам	по оборотам	1	2
ТаблицаПоОборотам (TurnTable)	остатки и обороты	по оборотам	8	7
ОстаткиПоПроводкам (TransSaldos)	начальный и конечный остаток, а также дебетовый и кредитовый оборот	по проводкам	4	3
ТаблицаПоПроводкам (TransTable)	проводки	по проводкам	4	6
Справочник (Reference)	имя справочника	по справочнику	1	1
Пусто (Empty)	пустая строка	любой	-	-

Секция **УточняемыеПараметры** используется только в том случае, если отчет строится в качестве уточняющего. При этом в данной секции перечисляется аналитика, которую уточняет отчет. Очевидно, что в уточняемом отчете должно быть разбиение по параметрам и/или корр. параметрам. Если разбиение выполнено по нескольким параметрам или есть разбиения по параметрам (корр. параметрам) по нескольким измерениям

(строки/столбцы/таблицы), то уточняемые значения перечисляются через запятую.

Секция **Период|Period** предназначена для вывода дат отчетного периода в различных форматах, что определяется макросом {Период}, которые используются в заголовке отчета секции с именем **Период|Period** и имеет следующий синтаксис:

```
$Макрос                = "{ "Период" [ ":" СтрокаФорматирования] }"
$СтрокаФорматирования = ФорматыВыводаДаты | "Прописью"
$ФорматыВыводаДаты     = "dd mmmm yyyy" | и т.д.
```

Для задания периода в общем виде используется макрос

```
За период: {Период}
```

Например:

```
За период: с 01.01.2007 по 31.12.2007
```

```
За период: с 01.01.2007 12:00:00 до 02.01.2007 12:00:00
```

Если требуется указать не порядковый номер месяца, а его полное название, например

```
За период: с 01 января 2007 по 31 декабря 2007
```

воспользуйтесь макросом

```
За период: {Период:dd mmmm yyyy}
```

Для задания произвольного периода прописью употребляется макрос вида:

```
За период: {Период:Прописью}
```

Например:

```
За период: 2007
```

```
За период: I квартал 2007
```

```
За период: с июня 2007 по август 2007
```

и т.д.

Клетки секций шаблона отчета на шаблоне можно форматировать произвольным образом: вводить текст, изменять цвет, выравнивание и т.д. Если текст не заключен в фигурные скобки, то он выводится, как есть. Если в клетке вставлены фигурные скобки (" ", без кавычек), то система попытается подобрать для отображения в этой клетке соответствующий (положению клетки) показатель из результатов отчета. Между фигурными скобками можно также писать текст, причем существует ограниченный набор ключевых слов (макросов), которые система "распознает", сопоставляя с ними лишь определенный тип показателей. Найдя макрос, система заменяет его содержимое на фактическое значение заданного показателя. Если макрос не найден, содержимое клетки считается комментарием – ни фигурные скобки, ни содержащийся в них текст в результирующей таблице не отображается, однако облегчает проектирование шаблона. Если структура отчета такова, что для этой клетки имеются данные в результатах отчета, то они будут выведены.

Системой обрабатываются следующие макросы:

Имя	Name
Описание	Description
ДатаНач	BegDate
ДатаКон	EndDate

Например, клетка со строкой "С ДатаНач по ДатаКон" будет, скорее всего, преобразована в строку "С XX.XX.XXXX по XX.XX.XXXX", где символами X помечены цифры фактических дат.

Вместе с тем, следует иметь в виду, что при неправильном использовании макросов (не по месту) возможны ситуации, когда для клетки с макросом не найдется соответствующего показателя в результатах отчета. Иными словами, указание какого-либо макроса в клетке не означает, что в ней обязательно будет выведен показатель, описываемый макросом – система может отобразить в клетке лишь те показатели, которые "ложатся" в эту клетку согласно общей структуре служебных секций. Если, например, в клетке указан макрос Имя, а в соответствии с текущим разбиением ей соответствует дата, то будет выведена именно дата, а не идентификатор чего бы то ни было.

Существует универсальный макрос {}, который означает вывод значений в соответствии с настройками отчета.

Если в клетке выводятся значения для нескольких параметров разбиения, то макросов в фигурных скобках может быть несколько (по числу параметров). Для удобочитаемости макросы могут отделяться друг от друга произвольными символами, например, запятыми (но не фигурными скобками).

В том случае, если разбиение на строки, столбцы или таблицы было задано для одного или нескольких параметров (набор параметров – это несколько параметров, указанных через запятую в поле **Параметр** на странице "Отчет" диалога настройки отчета), то в шаблоне можно конкретно указать, значения какого параметра (параметров) следует в данной клетке выводить.

Для этой цели в качестве макросов используются полные названия этих параметров, дополненные, при необходимости, цепочкой разыменований – правила записи таких макросов проще всего освоить, изучив в качестве примеров те строки, которые программа сама составляет в диалоге "Настройки разбиения по параметру" (на странице "Вывод").

Например, если разбиение идет по двум параметрам: Товар и Категория, причем параметр Товар является ссылочным, а Категория – просто число, то возможно составление макроса:

{Товар.Описание}, {Категория}

что предписывает программе вывести в клетке содержимое свойства **Описание** из элемента справочника товаров и категорию – товар и категория должны быть определены для данной клетки за счет соответствующего разбиения.

Для первого параметра разбиения, если он является ссылочным, возможно применение укороченного названия, включающего только имя свойства, т.е. без имени самого параметра разбиения. Так вышеприведенный макрос можно записать в краткой форме:

{Описание}, {Категория}

Если ссылочный параметр стоит не на первом месте в разбиении, то краткую форму использовать для него нельзя. Например, разбиение по паре тех же параметров, но в обратном порядке: Категория, Товар – имеет лишь один способ записи макроса (при условии, что нас интересует одно единственное свойство товара – описание):

{Категория}, {Товар.Описание}

Еще один пример для случая, когда в разбиении участвуют два ссылочных параметра: Товар и Склад. Здесь допустимы следующие эквивалентные макросы:

{Товар.Описание}, {Склад.Описание}

или

{Описание}, {Склад.Описание}

Свойства **Имя** и **Описание** применимы для клеток, в которых выводятся значения с разбиением по счетам и аналитическим параметрам.

Для аналитических параметров помимо имени и описания в клетку отчета можно выводить любой атрибут (поле аналитического справочника). Например, если отчет имеет разбиение на строки по товарам, а справочник товаров имеет поле Поставщик со ссылкой на контрагента, то в клетке первой колонки (по умолчанию, колонка Имя идет в шаблоне первой) можно ввести "Поставщик" для отображения информации о поставщике. Причем, если поле Поставщик в свою очередь имеет тип справочника, то можно его разыменовать для обращения к конкретному полю справочника поставщиков, например, "Поставщик.Адрес". Если же поле Адрес существует только в классе записи, послужившей источником информации для справочника поставщиков, но отсутствует в самом справочнике, то для обращения к этому полю необходимо предварить его имя символом решетки '#': "Поставщик.#Адрес".

Если шаблон используется в отчете по проводкам с разбиением на таблицы по документам или параметрам, то в секции **ЗаголовокТаблицы** доступны макросы по произвольному полю класса записи (в соответствии с MTL-описанием документа) с возможностью форматирования. Кроме того, поддерживается и разыменование ссылочных полей. При разбиении на таблицы по счетам в этой секции допустимы только макросы "Имя" и "Описание".

Формальный синтаксис заполнения клетки

Макрос	= "{ " [ТелоМакроса] " }"
ТелоМакроса	= [Содержимое][Количество] [Формат]
Содержимое	= ПоВремени ПоСчету ПоПараметру Функция
ПоСчету	= "Имя" "Описание"
ПоВремени	= "ДатаНач" "ДатаКон"
ПоПараметру	= [Атрибуты][. #Поля]
Атрибуты	= ИмяАтрибута[.ИмяАтрибута]
Поля	= ИмяПоля[.ИмяПоля]
Функция	= ИмяФункции "(" ПоПараметру ")"
ИмяФункции	= COUNT SUM MIN MAX AVERAGE
Формат	= ":" <строка форматного преобразования>
Количество	= "{ " "КолСтр" "КолКол" [":" <СтрокаФорматирования>] " }"

Внимание. В приведенном синтаксисе квадратными скобками обозначены необязательные компоненты.

Всё, что идет до первого '#' считается *полями справочника*, всё - что после, считается *полями записи*.

Например, если при разбиении по товарам каждый элемент в справочнике товаров имеет свойство Производитель, применим макрос:

{Производитель.#Регион.Название}

Здесь в клетку выводится название региона производителя (ссылка на регион имеется в поле записи о производителе, а Название - поле в записи о регионе: названия полей Регион и Название идут после '#').

Как видно из приведенного формального синтаксиса, в теле макроса может быть содержимое без указания формата (в этом случае вывод осуществляется с помощью формата по умолчанию) или формат без указания содержимого (в этом случае формат применяется для вывода значения разбиения по умолчанию - для простых параметров это значение параметра без каких-либо разыменований, функций и пр.), а также содержимое и формат.

Если последний идентификатор в череде разыменований соответствует полю справочника, то для него по умолчанию выводится описание (в справочнике всегда есть поле **Описание**). Если последний идентификатор является ссылочным полем записи, то в качестве выводимого значения берется **DocID**. Например, "{Товар.Производитель}" выведет описание производителя, а "{Товар.Производитель.#Регион}" - **DocID** записи о регионе.

Макросы для разбиений по времени ({ДатаНач} и {ДатаКон}) действуют только в тех ячейках отчета, куда должны выводиться значения разбиения по времени, а также в ячейках шапки отчета и строке отображения периода отчета. При этом макрос {ДатаКон} имеет смысл, только если в данную ячейку выводится период.

Макросы {ДатаНач} и {ДатаКон} **не действуют** при разбиении **по параметрам типа дата** - в этом случае действует синтаксис для параметров (т.е. варианты, обозначенные выше на схеме "ПоПараметрам" или "Функция").

Задание форматов клеток шаблона

В отчетах на шаблонах при наличии какого-либо произвольного текста в клетке шаблона, а также нескольких

макросов, формат клетки не меняется. Для показателей типа измеритель, частное, число, целое и дата в клетки шаблона помещается не статический текст, а соответствующее значение и требуемая строка форматирования, при условии наличия в клетке шаблона одного универсального макроса.

В отчетах по проводкам на шаблонах для показателей типа измеритель, частное, число, целое и дата также можно задавать форматы клеток шаблона.

Функции в теле макроса

В качестве функций в теле макроса можно применить одну из статистических функций, использующих в качестве аргумента параметр разбиения:

- **Count** - количество признаков;
- **Sum** - сумма значений свойства (только для типов Целое и Число);
- **Max** - максимальное значение свойства (только для типов Целое, Число, Строка и Дата);
- **Min** - минимальное значение свойства (только для типов Целое, Число, Строка и Дата);
- **Average** - среднее значение свойства (только для типов Целое и Число).

Эти функций также используются на странице "Вычисляемый" диалога ["Добавление показателя"](#).

Если с помощью макроса в клетке заказан вывод неких атрибутов, отсутствующих у фактически попадающих в эту клетку объектов, то система выводит в такую клетку сообщение об ошибке (уже в процессе построения отчета).

Макросы для вывода количества строк и колонок

Макросы {КолСтр} и {КолКол} предназначены для вывода количества строк и колонок, попавших в таблицу отчета по оборотам. Макросы применяются в клетках, в которых выводятся данные разбиения по строкам, таблицам и колонкам, а также в пользовательских столбцах. Макросы имеет смысл использовать в итоговых строках иерархии и в итоговой строке самой таблицы.

Примеры:

```
{КолСтр}  
{КолСтр: , #00}  
{КолКол}  
{КолСтр}, {КолКол}
```

Макросы для нумерации строк

В отчетах по оборотам предусмотрена возможность нумеровать строки, если в шаблон отчета введен макрос {ИндексСтр}, который имеет смысл в клетках, в которых выводятся данные разбиения по строкам, а также в пользовательских столбцах. В основных клетках разбиения, он используется совместно с универсальным макросом, например:

```
{ИндексСтр}. {}  
{ИндексСтр}) {}
```

Для формирования отчетов по оборотам на шаблонах используется секция **ТаблицаПоОборотам**. Столбцы этой секции могут иметь как зарезервированные имена, так и произвольные, а строки секции должны иметь только зарезервированные имена.

Секция **ТаблицаПоОборотам** используется также и для вывода результатов [отчета по справочнику](#).

Список столбцов и их назначение

Столбец	Назначение
1. РазбиениеПоТаблицам SplitByTable	разбиение по таблицам
2. Уточнение Inline	клетки, которые можно настроить в виде кнопок
3. Переключатель Switch	клетки, которые можно настроить в виде кнопок
4. РазбиениеПоСтрокам SplitByRow	разбиение по строкам
5. РазбиениеПоСтрокамДоп SplitByRowSep	задать стиль линии между столбцами
6. Столбец Split	виртуальный столбец
7. НачОстаток BegSaldo	
8. Оборот Turn	оборот
9. КонОстаток EndSaldo	конечный остаток
10. Дебет Debet	дебетовые показатели оборотов/остатков
11. Кредит Credit	кредитовые показатели оборотов/остатков

{ }					
{Нач. дата}		{Оборот}		{Кон. дата}	
{Дебет}	{Кредит}	{Дебет}	{Кредит}	{Дебет}	{Кредит}
{Изм}	{Изм}	{Изм}	{Изм}	{Изм}	{Изм}

Строка 1 указывается, если в отчете по оборотам требуется колонка для вывода разбиения по таблицам в случае "склеенных" таблиц. При отсутствии этой колонки формат вывода заголовков таблиц берется из секции с предопределенным именем [ЗаголовокТаблицы|TableHeader](#).

Строки 2 и 3 используются для задания столбцов, клетки, которых будут служить иконками, их можно настроить в виде кнопок. Эти кнопки будут использоваться для раскрытия интерактивного уточнения, а также раскрытия свернутых групп или их закрытия. Кнопка указывается для каждой строки, соответствующей группе.

Столбец 5 РазбиениеПоСтрокамДоп указывается сразу же после столбца 4 РазбиениеПоСтрокам и разрешает настраивать толщину вертикальных линий (толстая, тонкая или отсутствует) между колонками с выводимыми параметрами разбиения на строки. По умолчанию, если этого столбца нет, разделение столбцов выполняется тонкими линиями.

Строка 6 Столбец|Split предназначена для задания виртуального столбца, который может быть разделен на три или менее столбцов (НачОстаток, Оборот, КонОстаток), каждый из которых в свою очередь может быть также разделен на два столбца (Дебет и Кредит). Имена столбцов 6, 7 и 8 используются для вывода соответственно начальных остатков, оборотов и конечных остатков. Если в отчете необходимо выводить разделенные начальные остатки, то необходимо задать строку Дебет и Кредит. Таким образом, имя столбца, в который будут выводиться значения дебетового начального остатка, будет составным "Split.BegSaldo.Debet" или "Столбец.НачальныйОстаток.Дебет".

Порядок следования столбцов не регламентирован. Если какой-либо из стандартных столбцов необходимо не показывать в отчете, то его следует скрыть, а не удалять из секции.

Существует возможность добавления в секцию **ТаблицаПоОборотам** других столбцов с произвольным наполнением. При этом для их правильного форматирования и заполнения необходимо, чтобы строки секции были обязательно поименованы с использованием указанных предопределенных имен.

В [заполнении клеток](#) нестандартных столбцов могут участвовать обработчики события **ПриВыводе** или специальные макросы в фигурных скобках " ".

Список строк и их назначение

Строки секции должны иметь зарезервированные имена. Список строк приведен в той последовательности, которая представляется наиболее логичной, хотя теоретически порядок строк может быть иным:

Номер Имя

```

1      РазбиениеПоСтолбцам (SplitByCol)
2      ОстаткиОбороты (SumKind)
3      ДебетКредит (DebCre)
4      Измеритель (Measure)
5      ВходящийОстаток (TotalBegSaldoByRow)
6      Строка_Группа (Group)
7      Строка_СвернутаяГруппа (RollGroup)
8      СтрокаЧет_СвернутаяГруппа (RollGroupEven)
9      Строка_Элемент (NameByRow)
10     СтрокаЧет_Элемент (NameByEvenRow)
11     Строка_Итого (GroupTotal)
12     Строка_Группа_Уточнение (Inline_Group)
13     Строка_СвернутаяГруппа_Уточнение (Inline_RollGroup)
14     Строка_Элемент_Уточнение (Inline_NameByRow)
15     Строка_Итого_Уточнение (Inline_GroupTotal)
16     ИтогоПоСтрокам (TotalByRow)
17     ИтогоПоСтрокамВПроцентах (TotalPercentByRow)
18     ПрибыльПоСтрокам (ProfitByRow)
19     ИсходящийОстаток (TotalEndSaldoByRow)
20     ИтогоПоТаблицам (TotalByTable)

```

Строки с 1 по 4 формируют шапку таблицы.

Строки 5 и 19 содержат соответственно начальный и конечный остатки по всем столбцам разбиения. Эти строки показываются только в том случае, если в настройках отчета задано выводить обороты (свернутые или отдельные), так как начальные и конечные остатки показываются именно в этих столбцах.

Строка 6 содержит имя и наименование группы для иерархических отчетов. Перед названием групп (названием элементов групп и т.д.) можно указывать строку-сдвиг для выделения вложенных групп, используя макрос {Отступ}. Строка-сдвиг повторяется количество раз, равное степени вложенности группы. Для задания макроса используется следующий синтаксис:

```

$Макрос      = "{" ТелоМакроса "}"
$ТелоМакроса = "Отступ" [ ":" Строка ]
$Строка      = любая строка (считается, что в нее
                  входят все символы между ":" и "}" )

```

Если строка не указана, то по умолчанию она считается равной пробелу ' '.

Примеры:

```

{Отступ: }
{Отступ:>}
{Отступ}

```

Причем если группа свернута, то для неё можно настроить отдельную строку, и даже указать различное форматирование для четных и нечетных групповых строк с помощью строк Строка_СвернутаяГруппа и СтрокаЧет_СвернутаяГруппа. Если последняя строка не определена, то все свернутые групповые строки – и четные, и нечетные – выводятся с помощью строки Строка_СвернутаяГруппа.

Строки 9 и 10 предназначены для вывода данных из группы (эти строки "размножаются" в соответствии с количеством результатов в группе),

Для задания различных стилей для четных и нечетных строк в таблицу по оборотам нужно добавлять строку с именем:

```
СтрокаЧет_Элемент|NameByEvenRow
```

и, при желании, для свернутых групп:

```
СтрокаЧет_СвернутаяГруппа|RollGroupEven.
```

При отсутствии строки Строка_ЭлементЧет стили для четных строк берутся из строки Строка_Элемент. В иерархических отчетах отсчет строк (четная/нечетная) начинается в каждой группе заново. Если строки Строка_СвернутаяГруппаЧет в шаблоне нет, стили берутся из строки Строка_СвернутаяГруппа|RollGroup, а если и ее нет, то стили берутся в зависимости от четности из строк для элементов - Строка_Элемент, Строка_ЧетЭлемент.

В отчетах по оборотам можно управлять толщиной основных горизонтальных линий, например, в отчете "под зебру" горизонтальные линии можно вообще убрать. Толщина линий между строками отчета типа

```

"Строка_Элемент|NameByRow", "СтрокаЧет_Элемент|NameByEvenRow",
"Строка_СвернутаяГруппа|RollGroup", "СтрокаЧет_СвернутаяГруппа|RollGroupEven"

```

берется из настройки линии между строками шаблона

"Строка_Элемент|NameByRow" и "СтрокаЧет_Элемент|NameByEvenRow".

Строка 11 Строка_Итого используется для вывода итоговых показателей по группе, строка 16 ИтогоПоСтрокам – итогов по всей таблице.

Для задания формата строки Итого в %% в шаблоне заведена строка 17 с названием ИтогоПоСтрокамВПроцентах|TotalPercentByRow

Для строк, которые содержат интерактивное уточнение, в аналогичных случаях используются Строка_Группа_Уточнение, Строка_СвернутаяГруппа_Уточнение, Строка_Элемент_Уточнение, Строка_Итого_Уточнение. При необходимости итоги могут быть выведены в процентном отношении к общей величине показателя с помощью строки 17.

В таблице по оборотам строка 18 ПрибыльПоСтрокам|ProfitByRow применяется для задания формата строки Прибыль, в которой показывается прибыль, т.е. расхождение между дебетом и кредитом проводках (и в остатках и в оборотах). Строка отображается в отчете под строкой Итого, если в отчете настроен показ итоговой строки и хотя бы для одного показателя включен флаг **Вычислять прибыль** на странице ["Показатели"](#).

Для задания формата строки "Итого по отчету" в шаблоне используется строка 20 с названием ИтогоПоТаблицам|TotalByTable.

Предупреждение! При использовании старых вариантов шаблона формат данной строки берется из строки "Итого" (название ИтогоПоСтрокам|TotalByRow). Поэтому, если в первой клетке этой строки введен текст: Итого, его нужно заменить на {Итого} или просто {}. Строка "Итого по отчету" отображается, если установлен флаг **Итоговая строка по отчету** на странице ["Дополнительно"](#), а по каким показателям будут выводиться эти значения, определяется флагом **Показывать итог в %%** на странице "Показатели".

Для построения отчета по проводкам с использованием шаблона применяются секции:

- **ОстаткиПоПроводкам** - для вывода начального и конечного остатка, а также дебетового и кредитового оборота
- **ТаблицаПоПроводкам** - для вывода проводок.

Секция **ОстаткиПоПроводкам|TransSaldos** имеет четыре с предопределенными именами:

Номер Имя

1	НачальныйОстаток (BegSaldo)
2	ДебетовыйОборот (DebetTurn)
3	КредитовыйОборот (CreditTurn)
4	КонечныйОстаток (EndSaldo)

Три колонки этой секции используются для настройки вывода:

- заголовка, например: "Начальный остаток";
- имени измерителя: "Изм";
- значения остатка/оборота: " ".

При наличии только одного измерителя имя не выводится.

Имена столбцов в секции **ТаблицаПоПроводкам|TurnTable**:

Столбец	Назначение
Дата (Date)	дата проводки
Дебет (Debet)	счет дебета
Кредит (Credit)	счет кредита
Данные (Data)	показатель (сумма, комментарий и пр.) или счет, в зависимости от разбиения на колонки; данный столбец оставлен в целях сохранения совместимости со старым форматом шаблона; вместо данного столбца можно использовать следующие специализированные столбцы
Данные_Измеритель (Измеритель, Measure)	показатель, являющийся измерителем, т.е. сумма и признак
Данные_ИзмерительРасщ_Дебет (Data_MeasureEx_Debet)	аналогично Данные_Измеритель, но только для дебетовой составляющей (при разделении на дебет и кредит показателя)
Данные_ИзмерительРасщ_Кредит (Data_MeasureEx_Credit)	аналогично Данные_Измеритель, но только для кредитовой составляющей (при разделении на дебет и кредит показателя)
Данные_Аналитика (Аналитика, Analit)	клетки столбцов Данные_ИзмерительРасщ_Дебет и Данные_ИзмерительРасщ_Кредит можно объединять в строке Параметр, если формат вывода показателя предполагает наличие одного значения
Итого (Total)	показатель, являющийся аналитическим признаком
ПараметрыПроводки (Params)	нарастающий итог (виден, если в настройках отчета включен флаг Показывать итог по строке)
	параметры проводки, не вошедшие в число тех показателей, которые выводятся в собственных колонках (см. список Измерители/показатели в диалоге Внутренние отчеты)

Порядок следования столбцов в шаблоне не регламентирован. Некоторые столбцы могут автоматически скрываться системой в зависимости от разбиения. Например, при разбиении на таблицы по времени с шагом в 1 день колонка Дата скрывается.

При отсутствии столбцов Данные_ИзмерительРасщ_Дебет и Данные_ИзмерительРасщ_Кредит настройки берутся из столбца Данные_Измеритель, а если такого нет, то из столбца Данные.

Свойства видимости и выведения на печать столбцов шаблона (которые управляются флагами **Отображать на экране** и **Выводить на печать** диалога настройки свойств столбца) переносятся в соответствующие столбцы отчета.

Также при формировании отчета учитываются свойства строк исходного шаблона, то есть они переносятся в

бланк с результатами отчета. Сохраняются такие настройки как видимость (**Visible**) строки, ее вывод на печать (**Printed**) и необходимость начала новой страницы, начиная с этой строки. Кроме того, система берет из исходного шаблона такие свойства строк как способ определения высоты ("автоматически", "не менее, чем", "точно") и значение высоты в мм.

Строки в секции **ТаблицаПоПроводкам** также должны иметь предопределенные имена:

Номер Имя

1	Заголовок (Header)
2	Счет (Account)
3	Параметр (Param)
4	ПриходРасход (IncomeOutcome)
5	Проводка (Trans)
6	ПроводкаЧет TransEven
7	ИтогоПоИзмерителю (MeasureTotal)

Строки с 1 по 3 формируют шапку таблицы.

Строка 4 или 5 содержит сами данные (эта строка "размножается" в соответствии с количеством проводок). Строки 4 или 5 используются взаимозаменяемо в зависимости от того, включен ли отдельный вывод дебета и кредита (используется строка ПриходРасход) или нет (используется строка Проводка). Строка ИтогоПоИзмерителю используется для вывода итогов по измерителям. При отсутствии в шаблоне строк ПриходРасход и ИтогоПоИзмерителю настройки берутся из строк Параметр и Проводка.

В отчетах по проводкам разрешается использовать различные стили для выделения строк, например, четных и нечетных, или принадлежащих различным документам. Для этого в таблицу по проводкам нужно добавить строку с именем "ПроводкаЧет". При ее отсутствии стили для четных строк берутся из строки "Проводка".

Толщину основных горизонтальных линий в отчете можно изменять. Толщина линий берется из настройки линии между строками шаблона "Проводка|Trans" и "ПроводкаЧет|TransEven".

Как и в случае секции **ТаблицаПоОборотам**, если в секции **ТаблицаПоПроводкам** имеются нестандартные колонки, для их форматирования требуется, чтобы строки секции были поименованы. В противном случае, дополнительные столбцы выводятся с форматированием по умолчанию.

Структура учета – это средство организации учета на предприятии, которое формируется на стадии разработки проекта. Структура учета включает predetermined набор объектов учета, разрешенных Студией. Указанные в структуре учета объекты формируют учетную политику и определяют алгоритмическую реализацию функций проекта.

Объекты структуры учёта (справочники, журналы, счета и переменные) описываются по определенным правилам в одном или нескольких текстовых файлах с расширением *.lis, зарегистрированных в проекте Студии. В одном файле, который создается с помощью диалога "[Создание структуры учета](#)", разрешается описывать один или несколько объектов структуры учёта.

Описание каждого вида объекта структуры учета начинается с уникального ключевого слова, обеспечивающего идентификацию одного вида объектов от другого. Для обеспечения наглядности целесообразно снабжать описание объектов комментариями. В тексте описания большие и малые буквы не различаются.

Последовательность описания объектов в файле может быть произвольной. Объекты могут чередоваться друг с другом, причем, их число не ограничено. При этом, однако, следует иметь в виду, что файлы *.lis обрабатываются программой в порядке их перечисления. Если в описании сущности В есть ссылка на сущность А, то сущность А должна быть описана раньше либо в том же файле, что и В, либо в другом файле, указанном в проекте раньше, чем файл с описанием В.

Синтаксис описания каждого типа объекта структуры учета подробно рассматривается в следующих темах:

- [Аналитические справочники](#)
- [Обязательная аналитика](#)
- [Счета. Планы счетов](#)
- [Типы счетов](#)
- [Модификаторы параметров типов счетов](#)
- [Область учета](#)
- [Тип счета Перечисление|Enum](#)
- [Псевдонимы параметров типов счетов](#)
- [Описание журналов](#)
- [Описание журналов-картотек](#)
- [Описание переменных](#)

Назначение диалога: добавление нового уровня и настройка параметров [закрытия периода](#) под управлением Мастера.

На первой странице мастера необходимо задать название уровня. При проведении самой первой операции закрытия необходимо указать дату закрытия периода. При последующих операциях добавления уровня в поле **Период до** нужно проставить дату закрытия, которая не может быть позднее даты закрытия предыдущего уровня. Операция закрытия запрещает редактировать документы, но не экономит память и не ускоряет процесс обработки данных. Для уменьшения объема информации, хранимой в базе данных, следует выполнить свертку, установив флаг **Производить свертку**. Для перехода на следующую страницу нажмите кнопку **Далее**.

Внимание. При добавлении второго и последующих уровней флаг **Производить свертку** будет недоступен.

Вторая страница предназначена для задания параметров свертки. При установке флага **С учетом даты** или **С учетом документа** в сводную проводку объединяются проводки только с одинаковой датой или принадлежащие одному документу. Для параметров, перечисленных в списке, которые должны учитываться при свертке, необходимо установить флаг. Если справа от параметра установлен флаг, то по этому параметру сохраняется детализация. Наиболее плотная упаковка достигается при снятии всех флагов. Указав параметры свертки, нажмите кнопку **Далее**.

Третья страница позволяет проверить заданные параметры закрытия/свертки периода. Для проведения операции нажмите кнопку **Начать**. Если же параметры установлены неверно, то следует вернуться к предыдущей странице, нажав кнопку **Назад** и подкорректировать данные.

На последней странице в случае успешного завершения операции нажмите кнопку **Готово**. В результате в левой части окна при осуществлении первой операции появится папка **Закрытие** с одной операцией, вложенная в папку с текущей областью учета. Причем, если операция закрытие периода выполнена с использованием свертки, то она называется **Свертка**, иначе - **Закрытие**. При последующих добавлениях уровней появится папка **Свертка**. В каждую из этих папок добавляются операции по мере их проведения. При установке курсора в левой части окна на папке или операции в правой части окна отображается соответственно список выполненных операций или параметры операции **Закрытие** или **Свертка**.

Вызов диалога выполняется командой **Закрыть период (Ins)** или **Добавить уровень (Alt+Ins)** всплывающего меню из окна ["Управление закрытыми периодами"](#).

В структуре учета при описании параметров [типов счетов](#) могут использоваться следующие модификаторы:

- [Оборотный|Turn](#)
- [Индексируемый|Indexed](#)
- [НеИндексируемый|NotIndexed](#)
- [НеОкругляемый|UnRounded](#)
- [НеРазбиваемый|UnSplitted](#)
- [Накапливается|Accumulated](#)
- [НеНакапливается|NotAccumulated](#)

Оборотный|Turn

По умолчанию для каждого параметра счета подсчитываются как остатки, так и обороты. Если остатки не нужны, в описание параметра добавляется модификатор **Оборотный**. Если по параметру считаются остатки, то по умолчанию это происходит в режиме свернутого сальдо. Если необходимо получить разделенное сальдо, то в описание типа счета добавляется модификатор [Разделяющий](#), например:

```
тип Расчеты inherited Базовый;  
    параметр Контрагент      :спрКонтрагент, индексируемый;  
    параметр Сделка          :спрСделка;  
    параметр СчФактураВх     :спрСчФактураВх, оборотный;  
    параметр СчФактураИсх    :спрСчФактураИсх;  
    параметр Номер           :Целое = 1, индексируемый;  
    Разделяющий Контрагент, Сделка;  
конец;
```

Здесь все параметры имеют тип соответствующего [справочника](#), описанного в структуре учета, причем параметры (Контрагент и Сделка) будут использоваться для расчетов разделенного сальдо (для них нельзя указать модификатор **Оборотный**), а один параметр (СчФактураВх) - только для оборотов.

Следует иметь в виду, что указание того или иного модификатора ограничивает набор разрешенных операций для данного параметра. Так, по параметру, имеющему оборотный модификатор, нельзя получить остатки (будет возникать ошибка).

Индексируемый|Indexed

Модификатор **Индексируемый** предназначен для оптимизации (ускорения) отбора проводок/полупроводок по условиям отбора по параметрам. Для параметров, имеющих этот модификатор, система строит внутреннюю структуру данных (своего рода индекс), позволяющую сразу получать перечень всех проводок/полупроводок с одним и тем же значением заданного параметра. Индексируемые имеют смысл делать *лишь часто используемые в условиях отбора параметры*, причем чем больше будет предполагаемое разнообразие значений каждого такого параметра, тем менее эффективной окажется оптимизация, так как в пределе, когда каждое значение встречается в учетных данных лишь один раз, поддержка индексирования не даст никакого положительного эффекта, а лишь - накладные расходы (время, память компьютера).

Внимание. По умолчанию ссылочные параметры являются индексируемыми, а простые параметры - неиндексируемыми.

Неиндексируемый|Noindexed

Для аналитических параметров и измерителей внутренние структуры данных (индексы) ведутся по умолчанию. Атрибут **Индексируемый|Noindexed** применяется для отключения построения инвертированных списков (индексов), что позволяет экономить память, но может привести к замедлению выполнения различных запросов при отборе по этому параметру.

Примеры:

```
параметр СуммаНДС      :измеритель спрЕдИзм = nil, неиндексируемый;  
параметр СуммаУчетная :измеритель спрЕдИзм = nil, неокругляемый, неиндексируемый;
```

Неокругляемый|UnRounded

Модификатор **Неокругляемый** используется для параметров типа измеритель в тех случаях, когда их значения округлять не требуется, т.е. их значения сохраняются такими же, как они указываются при вводе проводки. По умолчанию параметры, не имеющие этого атрибута, округляются до точности валюты.

НеРазбиваемый|Unsplitted

Модификатор **Неразбиваемый** запрещает разбиение по параметру. Такие параметры исключаются из списков выбора аналитических разрезов в отчетах, то есть они не будут отображаться в списках параметров для разбиения по строкам, столбцам и таблицам в настройках встроенных отчетов, а также станут недоступны для установки разбиения через программный интерфейс.

Накапливается|Accumulated и Ненакапливается|NotAccumulated

Эти модификаторы определяют, следует ли системе строить аккумуляторы по данному параметру. По умолчанию в аккумуляторы включаются параметры, которые представляют собой измерители, аналитические параметры и перечисления. Параметры других типов по умолчанию не включаются в аккумуляторы. Указание модификатора **Накапливается** лишь разрешает использование аккумуляторов, однако не гарантирует, что они обязательно будут использоваться. При настройке [расчетной базы](#) на странице "Аккумуляторы" администратор имеет возможность включать и отключать аккумулятирование по каждому из параметров, который был описан с модификатором **Накапливается**. Ненакапливаемые параметры не показываются в этом диалоге.

Пример описания параметров:

```
параметр СуммаРасходов : измеритель спрВалюта = 0^руб, НЕнакапливается;  
параметр Товар : спрТовары, накапливается;
```

Программа обеспечивает комплексную автоматизацию разнообразных видов хозяйственной деятельности крупных предприятий самого разного профиля, что сопряжено с переработкой огромных объемов разнородных по своей структуре данных. Для повышения быстродействия обработки учетных движений целесообразно всю совокупность набора учетных данных разделить в соответствии со спецификой учета на логические области, называемые *областями учета*, которые будут обрабатываться [сервером расчетов](#) независимо друг от друга.

Итак, *область учета* - это независимое пространство проводок, отличающихся по своему составу и обрабатываемых раздельно. Следовательно, введение областей учета на сервере расчетов позволит не только разделить весь объем разнородных учетных данных на несколько независимых наборов, но и обрабатывать разные по составу проводки, относящиеся к разным видам учета хозяйственной деятельности (бухгалтерский, налоговый, управленческий и др.), параллельно и независимо друг от друга, исключив, таким образом, их нежелательное влияние друг на друга. К достоинству такого подхода относится тот факт, что, с одной стороны, ошибки, возникающие при обработке проводок из одной области учета, не влияют на обработку в другой области учета, а, с другой стороны, повышается скорость обработки пулов проводок.

Содержание темы:

- [Формальный синтаксис описания области учета](#)
- [Пример описания области учета](#)
- [Назначение атрибутов области учета](#)

Формальный синтаксис описания области учета

```
$ОписаниеОбласти = ( "Область" | "Domain" )
                    Имя области
                    [Синонимы]
                    [Атрибут]
                    [Название] ";"
$Синонимы          = ( "synonym" | "синоним" ) <Имя> { " , " <Имя> }
$Название          = ( "title" | "название" );
$Атрибут           = ( "Накапливается" | "Accumulated",
                    | "Ненакапливается" | "Notaccumulated",
                    | "Bytrans" | "ПоПроводкам",
                    | "Byhtrans" | "ПоПолупроводкам",
                    | "Closeable" | "Закрываемая",
                    | "Uncloseable" | "Незакрываемая" ) ";"
```

Область учета описывается в структуре учета в соответствии с приведенным синтаксисом в одном из текстовых файлах с расширением *.lis, которые [создаются](#) в редакторе проекта. Описание начинается с ключевого слова **Область|Domain**, за которым следует имя области и список атрибутов, разделенных запятыми.

Описывать область необязательно. Если область не задана, то по умолчанию план счетов и журналы приписываются к специальной области с именем *default*.

Область учета может включать один или [планов счетов](#), которые в структуре учета перечисляются после ее описания. Но каждый план счетов может относиться только к одной области, причем счета из различных областей учета, не могут корреспондировать друг с другом.

Журналы (и вместе с ними - проводки) приписываются к конкретной области учета путем указания ее имени и атрибутов перед описанием [свойств журнала](#) в структуре учета.

Пример описания области учета

```
область Склад accumulated, byhtrans, closeable;
счета Склад : Балансовый;
    счет Товары :ОстатокТовара;
    счет Расчеты :СальдоКорреспондента;
конец;

область Управление;
счета УправленческийПлан;
    счет Общий :Базовый;
конец;
```

Назначение атрибутов области учета

В структуре учета задаются predetermined значения атрибутов, которые можно изменить в диалоге [настроек расчетной базы](#). По умолчанию установлены следующие атрибуты: accumulated, byhtrans, uncloseable.

Атрибуты **Накапливается|Accumulated** и **Ненакапливается|NotAccumulated**

По умолчанию атрибут *Накапливается|Accumulated* установлен, т.е. по данной области учета разрешается строить аккумуляторы. Область не аккумулируется, если задан атрибут *Ненакапливается|NotAccumulated*, в этом случае аккумуляторы ни по каким параметрам не строятся, даже по тем, которые были описаны с модификатором *Накапливается* и используются по умолчанию.

Администратор имеет возможность на странице "Аккумуляторы" настроек [расчетной базы](#) включать и отключать аккумулирование по каждому из параметров, который был описан с модификатором *Накапливается*.

Атрибуты **Bytrans|ПоПроводкам** и **Byhtrans|ПоПолупроводкам**

По умолчанию аккумуляторы строятся по полупроводкам. Такие аккумуляторы, с одной стороны, будут более эффективны (у них выше кратность и меньше размер), но, с другой стороны, они менее применимы. Например, они не применяются во всех запросах к машине проводок, где используется корреспонденция счетов. Для обработки бухгалтерских проводок необходимо приписать к области учета атрибут *Bytrans|ПоПроводкам*, чтобы аккумуляторы строились по проводкам.

Атрибуты **Closeable|Закрываемая** и **Uncloseable|Незакрываемая**

По умолчанию, если эти атрибуты не заданы, область учета является незакрываемой, т.е. для нее запрещается операция [закрытия периодов](#). Чтобы сделать область учета "закрываемой", т.е. разрешить закрытие периода, следует установить атрибут *Closeable|Закрываемая*.

Обязательная аналитика - это аналитика, которая является частью проекта, т.е. необходима для реализации бухгалтерской логики в системе типовых операций. Примерами такой аналитики являются основные виды валют и единиц измерения, назначение платежа, виды нормируемых затрат.

Поскольку аналитика тесно связана со справочниками, синтаксис описания обязательной аналитики отражает данную связь и позволяет в случае необходимости объединить описание справочников и обязательной аналитики. Допускается также описание обязательной аналитики и в отдельных файлах, но при этом необходимо, чтобы идентификаторы справочника и соответствующей ему группы аналитических признаков совпадали.

Содержание темы:

- [Формальный синтаксис описания обязательной аналитики](#)
- [Примеры описания обязательной аналитики](#)
- [Описание обязательной аналитики в картотеке](#)

Формальный синтаксис описания обязательной аналитики

Для описания обязательной аналитики используется следующий синтаксис:

```
$ОбязательнаяАналитика      = ЗаголовокАналитики СписокЭлементов Конец ";"
$ЗаголовокАналитики          = ("Необходимо для"|"Necessary for")
                               ИмяСправочника
$СписокЭлементов              = { (ПростойЭлемент|Группа) }
$ПростойЭлемент               = ИмяЭлементаСправочника
                               [НаименованиеЭлемента]
$ИмяЭлементаСправочника      = Квалидент
$НаименованиеЭлемента         = ("Название"|"Title")
                               СтрокаВКавычках
$Группа                       = ("Группа"|"Group")
                               ИмяЭлементаСправочника
                               [НаименованиеЭлемента] ";"
                               СписокЭлементов
                               Конец ";"
$Конец                        = ("Конец"|"End")
```

Примеры описания обязательной аналитики

В соответствии с синтаксисом раздел описания обязательной аналитики начинается с ключевого слова **Необходимо для|Necessary for**, за которым следует имя справочника, например, **Необходимо для** спрНДС;. Такой синтаксис используется, когда описания обязательной аналитики и справочника находятся в разных файлах. Если же описание обязательной аналитики следует сразу за описанием справочника, имя справочника опускается и указывается только одно ключевое слово **Необходимо|Necessary**.

```
-- раздельное описание справочника и аналитики
Необходимо для спрНДС;
    0; -- не облагается налогом
    10; -- налог на НДС
    20;
Конец;
```

Внимание. В структуре учета при описании обязательной аналитики разрешается использовать идентификаторы, состоящие из цифр, при условии, что они обязательно описаны в справочниках.

```
-- совместное описание
Справочник ЕдИзм: изм title "Единица измерения";
Необходимо
    шт;
    кг;
    группа тара; -- разные ящики
    я1; -- Ящик малый
    я2; -- Ящик средний
Конец;
Конец;
```

Описание обязательной аналитики в картотеке

Важно понимать, что раздел обязательной аналитики сам по себе не описывает содержимое справочников, он только лишь позволяет указать компиляторам структуры учета и типовых операций обязательное наличие определенных элементов в определенных справочниках.

Внимание! Все обязательные аналитические признаки, описанные после ключевого слова **Необходимо для**, должны быть в обязательном порядке занесены в картотеки, являющиеся источниками элементов справочников. В результате этого обязательные признаки будут не только описаны, но также получат определенные значения для своих свойств (например, курс валюты).

При удалении элементов справочника они физически не удаляются, а помечаются как удаленные (потерянные). Как следствие, при удалении элемента справочника не потребуются пересвязывание или копирование, если в типовых операциях у него не было свойств, влияющих на связывание.

На сервере расчетов независимо от установки флага **Добавлять потерянную аналитику** в [настройках расчетной базы](#) запрещено добавление потерянной аналитики в справочниках, не имеющих привязки к документу.

В данной теме рассматриваются следующие вопросы:

- [Типы аналитических справочников](#)
- [Синтаксис описания справочников](#)
- [Пример описания справочника](#)
- [Свойства элементов справочников](#)
- [Свойства справочников единиц измерения](#)
- [Привязка к картотеке](#)
- [Незагружаемые свойства справочников](#)
- [Расширения справочников](#)

Типы аналитических справочников

Аналитические справочники предназначены для задания аналитических параметров счетов и единиц измерения. В Студии источниками признаков являются картотеки документов, которые рассматриваются как источники признаков (*элементов справочника*). По назначению аналитические справочники могут быть:

- по валютам;
- по единицам измерения;
- по аналитике общецелевого назначения.

Первые два типа являются специализированными, так как гарантируют наличие у элементов справочников особых свойств, характерных соответственно для описания валют (в частности, курсов) и единиц измерения. Эти два типа являются частными случаями аналитического справочника общецелевого назначения, так как фактически содержат специфическую аналитику, обработка которой реализована на системном уровне.

Синтаксис описания справочников

Описание справочника начинается с заголовка, идентифицируемого по ключевому слову **Справочник|Reference**, с несколькими обязательными и/или опциональными параметрами. Для [расширения](#) существующих справочников используется ключевое слово **расширяет|extends**. В заголовке справочника также указывается [привязка](#) к картотеке и записи. После заголовка может следовать описание дополнительных свойств, расширяющих набор свойств, неявно присутствующих в каждом справочнике. Завершается описание справочника ключевым словом **Конец|End**. Заголовок и каждый оператор в описании заканчивается символом ";" (точка с запятой). В любом месте описания можно использовать комментарий, начинающийся с двух символов "-" (тире или минус) и продолжающийся до конца строки. В целом данный синтаксис аналогичен синтаксису описания записей в языке MTL.

Формально синтаксис описания справочника можно определить следующим образом:

§

Пример описания справочника

```
-- справочник валют и единиц измерения
reference спрЕдИзм:Cur, from Справочники.ЕдиницаИзмерения,
card Справочники.картЕдиницаИзмерения, title 'Валюты и единицы измерения';
  field Код synonym Имя :String, from Код;
  field Наименование synonym
    Описание :String, notrefresh, from Имя;
  field Точность :Integer, from Точность;
  field Базовый :спрЕдИзм, from БазоваяЕдиница;
  field Множить :Logical, from Умножать;
  field Курс:Numeric period Date, from Курс;
  field ЭтоВалюта :Logical, from ЭтоВалюта;
Necessary
руб;
usd;
eur;
шт;
end;
```

Название справочника (title) может использоваться в качестве развернутого наименования справочника в диалогах ввода проводок и операций, поэтому рекомендуется формулировать название как существительное в единственном числе.

В заголовке справочника после ключевого слова **filter** можно задать логическое выражение (фильтр). В этом случае в справочнике будут отображаться только те записи, которые удовлетворяют установленному фильтру.

В описании справочника с помощью ключевого слова **Необходимо|Necessary** могут быть перечислены идентификаторы обязательных элементов справочника, что дает возможность уже на стадии компиляции проекта использовать данные идентификаторы в исходном коде классов и типовых операций. Раздел описания [обязательной аналитики](#) может быть отделен от описания справочников и находиться в другом файле, в этом случае после ключевого слова **Необходимо для** в явном виде указывается имя справочника, например, **Необходимо для** спрНДС;

Свойства элементов справочников

Каждый элемент справочника может иметь несколько свойств, которые используются во внутренних алгоритмах работы машины проводок и в условиях отбора полупроводок. Каждое свойство характеризуется типом и размерностью. Определяемые в описании дополнительные свойства элемента справочника могут иметь следующие типы: String | Integer | Numeric | Logical| <ИмяСправочника>. Последний тип означает, что свойство должно содержать ссылку на элемент указанного справочника, причем допустимо ссылаться на текущий справочник.

По размерности различаются скалярные свойства (например, ссылка на элемент справочника валют, задающий базовую валюту) или периодические свойства (например, курс валюты). Также как и неявно определённые свойства, дополнительные свойства могут быть как скалярными, так и периодическими величинами. В последнем случае после идентификатора типа должна следовать пара квадратных скобок "[]".

Каждый элемент любого справочника обязательно содержит три следующих свойства:

- **Name|Имя** - строка, однозначно идентифицирующая элемент справочника в текстовом источнике или условии отбора (квалифицированный идентификатор);

- **Description|Описание** - подробное описание элемента справочника;
- **Group|Группа** - Ссылка на элемент справочника – предок в иерархическом справочнике. У элементов, лежащих в корне, Group=nil.

Свойство элемента справочника **Необновляющий|NotRefresh** означает, что при его изменении машина проводок не будет пересвязывать журналы, а также запретит использовать это свойство при связывании (как в разнообразных условиях отбора, так и в коде операций). Пример использования свойства:

поле Описание: String, notrefresh;

Элементы справочников используются в типовых операциях, журналах, прикладных классах ТБ.Скрипт с синтаксисом, аналогичным использованию переменных соответствующих типов. Например, если описан справочник валют и параметры валют (например, usd) занесены в него после ключевого слова **Necessary** ([см. пример](#)), то с валютой можно работать программно примерно следующим образом:

СуммаВалUsd = СуммаРуб * Валюта.Usd.Курс[Today]

Здесь:

Валюта – имя справочника валют;

Usd – имя элемента справочника, определяющего иностранную валюту;

Курс – периодическое свойство элементов данного справочника, взятое на текущую дату.

При записи подобных выражений следует руководствоваться следующими правилами пересчета произвольной валюты в базовую:

```
If Валюта.Икс.Множить = True then
    СуммаВБазовойВалюте = СуммаВВалютеИкс * Валюта.Икс.Курс[Дата];
else
    СуммаВБазовойВалюте = СуммаВВалютеИкс / Валюта.Икс.Курс[Дата];
End;
```

Очевидно, что для обратного пересчета необходимо заменить операции умножения и деления на противоположные.

Следует иметь в виду, что в справочнике валют могут быть элементы, которые никак не соотнесены друг с другом с помощью поля Базовый. Более того, часть валют может иметь в качестве базовой одну валюту, а часть – другую. При этом кросскурсы таких не связанных между собой валют равны нулю.

Свойства справочников валют и единиц измерения

Для идентификации справочников валют и единиц измерения в заголовке справочника после двоеточия добавляются соответствующие ключевые слова Вал|Cur или Изм|Unit, ([см. пример](#)).

Внимание. Только один справочник может быть объявлен справочником валют.

Помимо обязательных свойств, имеющих у любого справочника, элементы справочников, задающих единицы измерения и валюты, имеют **дополнительные свойства**:

- **Base|Базовый** - ссылка на элемент справочника: базовую единицу измерения или валюту;
- **Rate|Курс** - скалярное или периодическое поле числового типа, определяющее отношение данной единицы измерения к базовой (формулу пересчета единиц см. ниже);
- **Mult|Множить** - логический признак деления или умножения на курс. Значение True означает, что для пересчета величин (**Vx**) в данной единице измерения в величины (**Vb**) в базовой единице необходимо их (**Vx**) умножать на курс, а значение False означает деление;
- **Accur|Точность** - целое число, задающее точность величины измерителя.

Привязка к картотеке

Аналитические справочники связываются с источниками значений (картотеками документов) с помощью ключевого слова **card**, после которого указывается полный путь к файлу *.cod с описанием картотеки в ветви Картотеки. С одним источником может быть связано несколько аналитических справочников.

Один аналитический справочник может быть *связан только с одним источником*. Каждому справочнику при этом сопоставляется имя класса записей (тип документов), из полей записи которого берутся значения для свойств элементов справочника. Класс записи (полный путь к файлу *.mtl в ветви Записи) указывается в заголовке справочника после ключевого слова **from**, например,

from Справочники.ЕдиницаИзмерения

Поля записей, из которых берутся значения для свойств справочника, также задаются с помощью ключевого слова **from**, например,

field Код :String, **from** DocId;

Незагружаемые свойства справочников

В целях экономии памяти в структуре учета разрешается некоторые свойства справочников, а также поля их структур, помечать как "незагружаемые" (виртуальные). Из предопределенных свойств "незагружаемым" может быть пока только "Описание" ("Наименование") элемента аналитики. Чтобы сделать поля "незагружаемыми", нужно в их описании указать атрибут **Незагружаемый | NotLoaded**, например:

```
reference sprУчетнаяЦена, from Данные.УчетнаяЦена, card Данные.картУчетнаяЦена;  
    field Наименование synonym Описание :String, notrefresh, notloaded;  
    ...  
end;
```

Значения свойств, помеченных как **NotLoaded**, не хранятся в памяти сервера расчетов. Однако, если они привязаны к полю документа, их можно использовать для вывода значений в отчет так же, как сейчас берутся поля записи, начинающиеся с символа #.

На эти свойства накладывается ряд ограничений. В частности, по таким свойствам нельзя делать условия отбора, использовать в разбиениях отчетов и обращаться к ним из типовых операций. Кроме этого, такие свойства исключаются из деревьев в мастерах условий отбора и выбора параметра разбиения.

Расширения справочников

Существует возможность не создавать новые справочники, а расширять существующие, описанные либо в этом же проекте ранее или в каком-либо из подключенных [подпроектов](#). Для этого необходимо в описании справочников перед словом **справочник|reference** вставить ключевое слово **Расширяет|Extends**. Примеры:

```
расширяет справочник Валюта;  
    поле СтранаВыпускаЕвро: Строка;  
конец;  
  
extends reference Корреспондент;  
    field Count1: Integer;  
    field Count2: Integer;  
end;
```

Поля, указанные в свойстве **Расширяет**, добавляются в указанный справочник. Новый справочник подобным описанием не создается.

В структуре учета описание картотечного журнала в [текстовом файле](#) с расширением *.lis начинается с заголовка журнала, в котором указываются имя и его назначение. Синтаксис [описания заголовка](#) такой же, как и для других журналов, за исключением следующего:

- для картотечных журналов тип не указывается;
- свойство **Version2|Версия2** используется только для описания картотечных журналов новой модели. Если оно не задано, то по умолчанию считается, что используется старая модель картотечных журналов;
- свойство **FreeTransEnter** добавляется в описание в том случае, когда разрешается вручную редактировать [свободные проводки](#), которые автоматически сгенерированы типовыми операциями. При отсутствии этого свойства редактировать проводки запрещено.

Внимание. Свойства, указанные в заголовке журнала отделяются запятыми, а после последнего из них ставится ";".

Пример описания нового картотечного журнала

```
журнал Движения title "Учетные движения"
:freetransenter, version2;
Priority 0; -- Приоритет журнала
Records Управление.Данные.Процесс;
Card Управление.Данные.картПроцесс;
Filter "not НеПорождаетПроводки and (ДатаНачала <> nil)";
DateField ДатаНачала;
EnableField Проведен;
PriorityField Номер;
LoadingFields
    ТипПроцесса, Спецификация, DocId, УровеньКонфиденциальности,
    Позиции.КоличКуда, Позиции.КоличОткуда, Позиции.Колич2Куда,
    ....
Operation ОперДвижение.Провести;
end;
```

Описания свойств картотечного журнала

Основным признаком, показывающим, что используется новая модель журналов, является наличие свойства **Version2** (вторая версия) в заголовке журнала. Отличительной особенностью новой модели от старой является отсутствие привязки полей документа к типовой операции, потому что привязка полей перенесена в саму типовую операцию.

В новой модели картотечных журналов могут использоваться следующие свойства, которые в соответствии с синтаксисом описываются после заголовка журнала:

- **Records** - обязательное свойство, задает доступную в проекте запись (документ), служащую источником данных для журнала. Для гетерогенных журналов записей может быть несколько, все они перечисляются через запятую;
- **Card** - необязательное свойство, используется для ввода имени картотеки, описанной в проекте для записи;
- **DateField** - обязательное свойство, задает поле типа **Дата**, содержащего дату документа;
- **Filter** - необязательное свойство, для установки логического выражения (фильтра). В журнале отображаются только те записи, для которых выражение истинно;
- **PriorityField** - необязательное свойство, служит для задания поля, содержащего значение приоритета. Если в картотеке окажется несколько записей с одинаковыми наборами прочих используемых полей, данное поле позволяет определить порядок, в котором эти записи будут формировать проводки;
- **EnableField** - необязательное свойство, задает логическое поле, на основании которого определяется, следует ли проводить (учитывать в расчетах) конкретный документ картотеки;
- **LoadingFields** - обязательное свойство, для перечисления загружаемых полей, которые отделяются друг от друга запятыми;
- **Operation** - обязательное свойство, вводит названия типовых операций, зарегистрированных в проекте, которые будут вызываться для каждого документа и формировать на его основе проводки.

Внимание. Операция допустима в журнале, если в ее заголовке указан тот же класс документа, что и в журнале, либо базовый класс для всех документов журнала.

Синтаксис свойств для расширения журналов

Свойство **Extends** позволяет изменять алгоритмы проведения в [надпроектах](#), при условии, что в классе типовой операции имеется наследник, в котором меняется алгоритм проведения. Эта новая типовая операция используется вместо старой. Синтаксис свойства **Extends**, используемого для расширения журналов, аналогичен описанию свойств журнала:

```
Extends |Расширяет Journal |Журнал <ИмяЖурнала> [Синонимы] [Название];  
    <Список свойств журнала>  
End |Конец;
```

Например:

```
extends journal    Отгрузка;  
    Operation ОперДвижение.Провести1;  
end;
```

Отличие от расширений справочников и типов счетов состоит в том, что только свойство **LoadingFields** добавляется к **LoadingFields** базового журнала, а остальные свойства, если они установлены в свойстве **Extends**, замещают значение свойства базового журнала. В частности, если **Extends** содержит операции, то все операции базового журнала отключаются и вместо них используются операции, описанные в свойстве **Extends**.

В программе существует особый механизм, предназначенный для хранения нескольких значений одного и того же параметра, изменяющегося с течением времени. Каждый такой параметр называется *периодической переменной*.

Переменные - это именованные сущности, имеющие (обычно) периодические значения простого типа и предназначенные для управления бухгалтерскими расчетами. В качестве примеров переменных можно привести разнообразные нормы отчисления и налогообложения. Вне зависимости от того, как описана переменная - как скалярная или периодическая - в типовых операциях она видна как скалярная величина со значением, определенным на дату выполнения типовой операции.

Периодические (общие) переменные, используемые на данном участке бухгалтерского учета, описываются в структуре учета вручную в соответствии с синтаксисом языка ТБ.Скрипт в текстовых файлах с расширением *.lis. Наряду с общими переменными в этих файлах также могут описываться и другие объекты структуры учета, например, журналы и справочники.

Синтаксис описания переменных описывается следующим образом:

\$

Пример:

Для доступа к переменной из ТБ.Скрипт необходимо использовать функцию GetVariable. Например, для конструирования измерителя с использованием переменной ВалютаУчета, можно применить следующую запись:

Использование переменных

Периодические переменные используются по аналогии с обычными переменными с той лишь разницей, что они определены в контексте всего проекта. То есть, если в структуре учета описана, например, переменная-массив с идентификатором СтавкаНалогаСПродаж:

то в любом модуле (COD-файле) проекта можно получить ее значение на конкретную дату:

Если требуется получить значение периодической переменной на текущую дату, то квадратные скобки и индекс можно опустить, так как запись:

эквивалентна:

где Сегодня - это системная функция, возвращающая текущую дату.

В случае, если общая переменная не является многозначной, то есть описана в структуре учета со скалярным типом (без квадратных скобок), ее использование полностью совпадает по синтаксису с обращением к простой переменной - достаточно написать в выражении ее идентификатор.

В структуре учета под псевдонимом параметра понимается **новый виртуальный параметр типа счета, по значению всегда равный базовому параметру**. При использовании псевдонима синтаксис описания параметров **типов счетов** имеет следующий вид:

```
$ОписаниеПсевдонима = "Parameter|Параметр" <Имя> [Синонимы] "Equal|Равен"  
                      <ИмяБазовогоПараметра> [Название] ";"  
$Синонимы             = "Synonym|Синоним" <Имя> {"", " <Имя>}"  
$Название             = "Title|Название" <Название>
```

Где

<ИмяБазовогоПараметра> - имя параметра, который должен быть описан ранее в данном типе счета или унаследован, и не является псевдонимом.

Как видно из описания синтаксиса псевдонимы параметров могут иметь синонимы и заголовок, но недопустимо использование ключевых слов типа indexed, accumulated и других атрибутов.

Пример:

```
Parameter Банк synonym Bank equal Процесс title "Банк Авангард";
```

На псевдонимы параметров накладываются те же самые ограничения, что и на обычные параметры, но главное, чтобы **их описание не отличалось для разных типов счетов**. То есть, если в одном типе счета описан псевдоним S1 equal P1, то в другом уже нельзя написать что S1 equal P2. У одного параметра может быть несколько псевдонимов. Они наследуются по иерархии счетов. В наследниках к параметрам, описанным в базовых классах, могут быть приписаны новые псевдонимы.

В условиях отбора псевдоним ведет себя следующим образом. Например, если имеется описание вида:

```
parameter S1 equal P1;
```

то условию типа

```
S1 = X
```

удовлетворяют все проводки, в которых есть параметр с данным псевдонимом, и значение этого параметра равно X, А условию

```
S1 <> X
```

удовлетворяют *все остальные проводки*, в которых:

- параметр P1 переименован в S1, но не равен X;
- параметр P1 не переименован в S1, пусть даже он и равен X;
- вообще нет параметра P1.

Описание или удаление нового псевдонима не приводит к пересвязыванию проводок. Операции чувствительны к изменениям псевдонимов только синтаксически, если удален псевдоним, использовавшийся в операциях, то они не откомпилируются, их нужно будет исправить, и в результате этого произойдет пересвязывание. Если же псевдонимы не использовались в операциях, или изменились так, что синтаксически их изменение выявить невозможно, то операции остаются неизменными. Программист может, если он считает нужным, выполнить полную перекомпиляцию проектов.

Псевдонимы можно использовать в типовых операциях, в текстовых журналах, в табличных журналах (ввод проводки), в табличных и картотечных журналах при показе проводок. А также в оборотных отчетах, в отчетах по проводкам (разбиения, показатели, мастера фильтров), в условиях отбора, в машине проводок (псевдонимы параметров используют индексы и аккумуляторы базовых параметров), в просмотре структуры учета.

Введенные в структуру учета псевдонимы параметров регистрируются в **классах полупроводок** наряду с обычными параметрами и их можно использовать в типовых операциях для установки значений соответствующих параметров.

Все типы [журналов](#) добавляются в проект на стадии его разработки. Свойства журнала зависят от его типа, который указывается вручную в структуре учета при описании журнала в текстовых файлах с расширением *.lis. В одном файле, который создается с помощью диалога "[Создание структуры учета](#)", разрешается описывать один или несколько журналов, а также другие объекты структуры учета (справочники, счета и переменные).

После создания файла *.lis можно приступить к его редактированию. Для добавления нового журнала следует описать его в соответствии с синтаксисом языка ТБ.Скрипт. Описание журнала начинается с заголовка журнала, за которым следуют свойства журнала, заканчивается описание ключевым словом **End**.

Содержание темы:

[Описание заголовка журнала](#)
[Примеры описания текстового и табличного журналов](#)
[Описание свойств текстового и табличного журнала](#)
[Описание области учета](#)

Описание заголовка журнала

Для всех трех типов журналов заголовок журнала начинается с ключевого слова **Journal|Журнал**, за которым следует его имя и назначение. После знака ":" описания журналов различаются. Так, для текстовых и табличных журналов после двоеточия указывается их тип, а для картотечных журналов - необязательный номер версии и возможность ввода свободных проводок.

Для описания заголовка нового типа журнала используется следующий синтаксис:

- **Journal|Журнал** - обязательное ключевое слово;
- **Имя** - обязательное имя журнала, произвольная строка, удовлетворяющая требованиям составления идентификаторов. Имя журнала должно быть уникальным в контексте всего проекта, с помощью которого происходит обращение к журналу и под которым журнал регистрируется в проекте и отображается в сессии в [списке](#) журналов;
- **Title|Название** - необязательное ключевое слово, используется в тех случаях, когда за ним следует описание назначения журнала;
- **Назначение** - необязательное описание назначения журнала, используется только совместно с ключевым словом **Title|Название**, иначе - выдается сообщение об ошибке;
- **:** - двоеточие ":", за которым следует тип журнала;
- **Text|Table** - конструкция используется для идентификации типов журналов. Для текстовых журналов указывается ключевое слово Text, для табличных - Table. Для картотечных журналов тип не указывается, т.е. в их описании отсутствуют ключевые слова Text или Table;
- **Version2|Версия2** - необязательное свойство, характеризующее номер версии журнала, используется только для описания картотечных журналов новой модели. Если эта конструкция не задана, то по умолчанию считается, что используется старая модель картотечных журналов;
- **FreeTransEnter** - необязательное свойство, разрешающее вручную редактировать свободные проводки, которые автоматически сгенерированы типовыми операциями, При отсутствии свойства в описании редактировать проводки запрещено.

Внимание. Свойства, указанные в заголовке журнала отделяются запятыми, а после последнего из них ставится ";".

Примеры описания текстового и табличного журналов

```
Журнал Журнал1 title "Текстовый журнал по бухгалтерии1" :Text;  
Priority 0;  
end;
```

```
Журнал ТабЖур1 title "Табличный журнал по бухгалтерии" :Table;  
Priority 0;  
Card ".Журналы.ТабЖур1";  
end;
```

Пример картотечного журнала приведен в теме "[Описание картотечного журнала](#)".

Описание свойств текстового и табличного журнала

После компиляции проекта все произведенные изменения в файле вступают в силу.

В зависимости от выбранного типа журнала, его свойства задаются по-разному. Однако свойство **Priority**

присуще всем журналам. В тех случаях, когда важен порядок обработки операций из разных журналов за одну дату, необходимо в описании журнала задать свойство

```
Priority 0;
```

Чем меньше значение числа, которое указывается за словом **Priority**, тем раньше обрабатывается данная операция. Проводки, устанавливающие курсы валют, должны обрабатываться в первую очередь и иметь нулевой приоритет.

Следует заметить, что в табличных журналах имеется столбец **Приоритет**, в котором указывается приоритет обработки операции для каждой записи. В том случае, когда записи за одну дату имеют одинаковый приоритет, в первую очередь будет обрабатываться запись из журнала, имеющего меньший приоритет.

Внимание. Текстовые журналы не имеют никаких свойств, кроме **Priority**.

Табличные журналы имеют дополнительно возможность выбрать картотеку, в окне которой следует отображать содержимое данного конкретного журнала. Для этой цели используется свойство

```
Card ".Журналы.ТабЖур1";
```

Привязка табличного журнала к картотеке не обязательна. Если в описании табличного журнала это свойство отсутствует, система будет отображать содержимое журнала в стандартном окне. В этом случае выводятся столбцы: Дата, Операция, Сумма, Комментарий.

Описание области учета

Все используемые в проекте журналы должны быть распределены по существующим областям учета. Для этого перед описанием журнала в файле *.lis должна стоять конструкция, предназначенная для ввода идентификатора имени области учета, вида:

```
область Раздел_Баланс accumulated, bytrans, closeable;
```

[Область учета](#) определяет логическую совокупность набора учетных данных, которая будет обрабатываться [сервером расчетов](#) независимо от других областей. Каждый журнал может принадлежать только одной области учета. Каждая область учета может содержать несколько журналов. Планы счетов, описанные в структуре учета, также распределяются по областям. Область учета журнала может быть не указана - в этом случае система размещает журнал в специальной области с именем *default*.

План счетов представляет собой перечень счетов, объединенных по какому-либо признаку, например, по назначению. Планов счетов может быть несколько, но каждый счет относится только к одному плану счетов и характеризуется уникальным именем и типом счета. Имя счета представляет собой квалиидент (короткий идентификатор). В отличие от других сущностей программы, имя счета может начинаться с цифры. Имя счета должно быть уникальным во всей структуре учета, то есть в разных планах не могут встречаться одноименные счета.

Предупреждение. Описание типов счетов *должно предшествовать их использованию в плане счетов*.

Типы счетов имеют формальные параметры, но могут не иметь ни одного параметра. При использовании счетов в проводках и полупроводках указываются фактические значения этих параметров.

Содержание темы:

- [Описание плана счетов](#)
- [Область учета](#)
- [Описание счета или субсчета](#)
- [Общий пример плана счетов](#)

Описание плана счетов

План счетов описывается в [структуре учета](#) в файле с расширением *.lis. Описание начинается с ключевого слова **Счета|Accounts**, за которым следует уникальное имя плана счетов (идентификатор). Если есть необходимость описать счета одного плана счетов в разных файлах, то необходимо повторить параметры плана счетов в каждом файле. В любом случае первое по порядку компиляции описание плана счетов определяет свойства плана счетов. Повторные писания присоединяются к первому, требуется только их непротиворечивость.

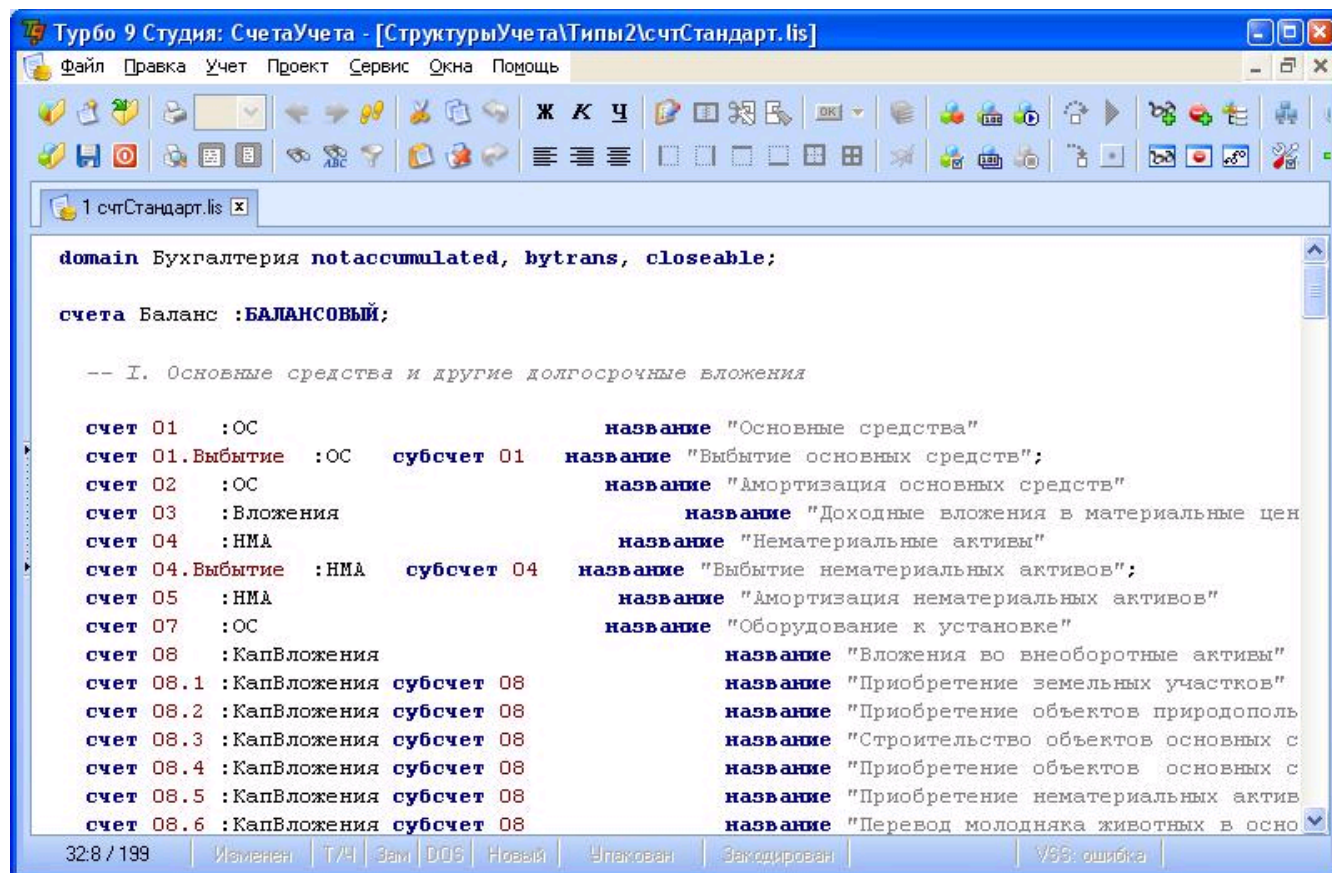


Рис. План счетов.

Раздел описания плана счетов может содержать несколько частей, в которых определяются:

- типы счетов (эта часть может быть вынесена за план счетов);
- счета;
- аккумуляторы, используемые в данном плане счетов.

План счетов описывается в соответствии со следующим синтаксисом:

```
$ПланСчетов          = ЗаголовокПланаСчетов
                      {ЭлементПланаСчетов} Конец ";" .
$ЗаголовокПланаСчетов = ("Счета"|"Accounts")
                      ИмяПланаСчетов [Балансовый] ";" .
$ИмяПланаСчетов       = Идентификатор .
$Балансовый           = ":" ("Балансовый"|"Balanced").
$ЭлементПланаСчетов   = ([ ОписаниеТипа
                        |ОписаниеСчета
                        |ОписаниеСубсчета
                        |ОписаниеАккумулятора) .
$Конец                = ("Конец"|"End") .
```

Наличие ключевого слова **Балансовый|Balanced** в описании заголовка плана счетов означает, что план счетов является балансовым, т.е. по счетам этого плана разрешается *формировать только проводки* и не допускается - полупроводки.

Если проект использует подпроекты, существует возможность добавлять счета в существующие в подпроектах планы счетов, а не в новые. Тем самым происходит расширение описания исходного плана. Для этого достаточно описать расширяемый план счетов повторно в рабочем проекте (который использует подпроект, где указанный план был описан впервые) и внести в него добавляемые счета.

Область учета

Для оптимизации скорости обработки пулов проводок, относящихся к различным планам счетов, программа позволяет разносить планы счетов по различным [областям учета](#). Каждая область может включать один план счетов или несколько, но конкретный *план счетов может относиться только к одной области*.

Указание области является необязательным. Если область не задана, то план счетов по умолчанию привязывается к специальной области с именем *default*.

Описание счета или субсчета

```
$ОписаниеСчета        = [ПризнакКонсолидации]
                      ("Счет"|"Асс") ИмяСчета ":" ИмяТипа [Название]
$ПризнакКонсолидации   = "Объединяющий" | "Consolidating"
$ОписаниеСубсчета      = ("Счет"|"Асс") ИмяСубсчета
                      "Субсчет" | "Subacc" ИмяСчета [Название]
$ИмяСчета              = Квалифицированный идентификатор
```

Некоторые счета могут рассматриваться как составные сущности, имеющие несколько подчиненных отделов, называемых субсчетами. Субсчет является полноценным счетом. Допускается произвольная вложенность субсчетов, то есть субсчет может также содержать субсчета более нижнего уровня.

Счета, помеченные как объединяющие (ключевое слова **Объединяющий|Consolidating**), не могут иметь разделяющие параметры, т.е. в описании типа такого счета не должно быть параметров с атрибутом **Разделяющий|Separator**. Если для счета задан атрибут **объединяющий/consolidating**, то при построении иерархических отчетов по счетам, сальдо по этому счету будет свернутым.

Для привязки субсчета к вышестоящему счету в описании плана счетов используется ключевое слово **Субсчет|Subacc**.

(

Пример:

Идентификатор счета может иметь в своем составе точку, которая трактуется как обычный символ. Иными словами, счет с именем 68.НДС не является обязательно субсчетом счета 68 лишь на основании того, что у них присутствует общий корень в идентификаторе. При использовании субсчетов в типовых операциях и проводках вводится указанный в плане счетов простой идентификатор. Субсчета должны описываться после "родительских" счетов.

Привязка субсчета к вышестоящему счету по умолчанию осуществляется в режиме разделенного сальдо. Если по субсчетам некоторого счета не нужно разделенное сальдо, то описание счета предваряется модификатором **Объединяющий (Consolidating)**. Пример:

Здесь счет 68 описывается таким образом, что по его субсчетам будет подсчитываться свернутое сальдо.

Запрещение разделенного сальдо позволяет уменьшить накладные расходы ресурсов компьютера.

При необходимости разработчик может разрешить пользователю создавать новые счета непосредственно во время выполнения проекта. Такие счета называются дополнительными и, как правило, должны быть субсчетами каких-либо стандартных счетов, описанных в структуре учета проекта. Для того чтобы дать пользователю возможность формировать дополнительные счета, необходимо создать в проекте картотеку для записи Kernel.Счета и добавить в нее столбцы:

Обработка дополнительных счетов считается самым первым этапом в обработке данных, таким образом ошибка в дополнительных счетах полностью парализует работу машины проводок. Любое изменение дополнительных счетов сейчас приводит к полной перекомпиляции журналов и их пересвязыванию.

Общий пример плана счетов

```
СЧЕТА Баланс;
-- План счетов торгово-складского учета --
тип Базовый; -- только сумма
    параметр Сумма: измеритель спрВал;
    -- Параметр Сумма явно определен как измеритель,
    -- в противном случае получим аналитический
    -- параметр по справочнику валют.
    параметр Комментарий: string = nil;
конец;

тип ССубДвиж Унаследованный Базовый;
    параметр Суб : Суб = nil;
    параметр Движ : string = nil;
    параметр Юл : Юл = nil;
    параметр К : К = nil;
    параметр ТСд : string = nil;
    параметр УчДолг: string = nil;
конец;

type СМесто Унаследованный ССубДвиж;
    parameter Место: Место = nil;
end;

тип СРес Унаследованный СМесто;
    параметр Рес : Рес = nil;
    параметр ЕДИзм: измеритель ЕДИзм = nil;
конец;

тип СДеньги Унаследованный СРес;
    параметр Деньги : Деньги = nil;
конец;

тип СФ Унаследованный ССубДвиж;
    параметр Ф : Ф = nil;
конец;

счет 0.0 :ССубДвиж название
    "Денежные средства, краткосрочные денежные обязательства";
    -- Материальные активы (ТМЦ)
    -- учет в деньгах и в количестве
счет 0.37 :СРес название "Выпуск продукции";
счет 0.40 :СРес название "Ресурсы";
счет 0.45 :ССубДвиж название "Ресурсы, переданные на реализацию";
счет 0.46 :СРес название "Реализация товаров и услуг";
счет 0.50 :СДеньги название "Деньги";
счет 0.52 :СДеньги название "Транзитный для денег";
счет 1.62 :СДеньги название "Покупатели";
конец;
```

В табличный журнал (ТЖ) заносятся операции двух типов - типовые и свободные. Типовая операция - алгоритм на языке ТБ.Скрипт, выполняемый на сервере расчетов и генерирующий проводки и полупроводки. Алгоритм описывается в файлах с расширением .cod в ветке "Учетные операции" окна [редактора проекта](#). Единичные проводки в ТЖ можно заносить только в составе свободной операции.

Типовая операция генерирует проводки в соответствии со своим описанием в cod-файле. *Свободная операция* не требует предварительного описания, ее проводки вводятся непосредственно в ТЖ одновременно с вводом операции. Типовая операция может быть в любой момент превращена в свободную, при этом в ТЖ будут занесены ее проводки, которые затем могут быть отредактированы. При таком раскрытии, сохраняется сама операция, набор ее параметров, и проводки сохраняют с ней связь. Раскрытая типовая операция становится доступной только на чтение, т.е. ее нельзя редактировать как операцию (нельзя менять значения ее параметров), а можно только править ее проводки. Раскрытая типовая операция может быть снова "закрыта", при этом все ее проводки удаляются из ТЖ, и она снова начинает работать в соответствии со своим описанием.

Физически один ТЖ представляет собой группу таблиц, которая создается в момент создания [информационной базы](#) наравне с таблицами всех классов документов, но, в отличие от них, его описание на языке MTL не требуется.

Табличный журнал добавляется в проект на стадии его разработки после [создания текстового файла](#) с расширением *.lis и вставки в него [описания свойств](#) табличного журнала.

Перечислимые типы описываются в структуре учёта внутри [типов счетов](#) и во многом схожи с перечислениями ТБ.Скрипт. Перечислимый тип предназначен для автоматизации ввода. Элементы перечисления можно рассматривать как целые константы. По умолчанию их значения представляют собой последовательность натуральных чисел, но при необходимости их можно переопределить, причем при описании элементов их значения можно задать любыми и в произвольном порядке.

Синтаксис описания перечисления следующий:

```
$ОписаниеПеречисления = ( "Enum" | "Перечисление" ) ИмяПеречисления  
                        [Синонимы] [Название] ";"  
                        {ЭлементПеречисления ";" }  
                        ( "End" | "Конец" ) ";"  
$Синонимы              = ( "Synonym" | "Синоним" )  
                        ИмяПеречисления { " ," ИмяПеречисления }  
$ИмяПеречисления      = Идентификатор  
$Название              = ( "Title" | "Название" ) строка  
$ЭлементПеречисления  = ИмяЭлемента [Синонимы] [Значение] [Название] ";"  
$Значение              = "=" целое число
```

Описание перечисления начинается со слова **Enum|Перечисление** и заканчивается словом **End|Конец**. Как видно из вышеприведенной записи, обязательными частями описания перечисления являются идентификаторы самого перечисления и его элементов. Все остальные части - синонимы, названия и целочисленные значения элементов - указываются опционально.

Пример:

```
type Базовый;  
  enum ТТипОплаты title "Способ расстаться с денежками";  
    Cash      synonym Нал, Деньги title "Наличные";  
    Clearing  synonym Безнал      title "Безналичные";  
    Barter    synonym Бартер      = 4 title "Бартер";  
    Giro      synonym Долг        title "Взаимозачет";  
  end ;  
  ...  
end  
  
тип Базовый synonym Fundamental;  
  
  enum ТипПлатежа synonym PayType;  
    Расчет   = 0 title "Взаиморасчеты";  
    Налоги   = 1;  
    Реклама  = 2;  
    Взятка   = 3;  
  end;
```

Если значение элемента перечисления опущено, то его значение по умолчанию берется на 1 большим, чем у предыдущего элемента, причем первый элемент по умолчанию имеет значение 1.

После компиляции структуры учета перечисления отображаются в иерархии классов ТБ.Скрипт как пользовательские типы, описанные внутри классов - типов счетов.

Параметры типов счетов типа перечисление описываются следующим образом:

```
parameter ТипОплаты : ТТипОплаты = Cash;
```

либо, если параметр описывается вне типа счета хранящего перечисление:

```
parameter ТипОплаты : Базовый.ТТипОплаты = Cash ;
```


Каждый счет, описанный в [плане счетов](#), должен иметь уникальное имя и тип счета, который характеризует принадлежность счета к тому или иному виду учета, например, к бухгалтерскому, управленческому или другому виду. Описание типов счетов *должно предшествовать их использованию в плане счетов*. Рекомендуется типы счетов в [структуре учета](#) записывать перед планом счетов, использующим счета этих типов.

Тип счета обозначается простым идентификатором и может быть описан как расширение другого типа счета (см. ниже). Типы счетов имеют формальные параметры, но могут и не иметь ни одного параметра. Если в двух разных типах счетов есть одноименные параметры, то их типы должны совпадать. Каждый тип счета имеет свой уникальный набор параметров.

В данной теме рассматриваются следующие вопросы, характеризующие правила использования типов счетов в программе:

- [Формальный синтаксис описание типа счетов и их параметров](#)
- [Описание типа счета. Модификатор Разделяющий| Separator](#)
- [Описание параметров типов счетов](#)
- [Наследование типа счета](#)
- [Расширение типа счета](#)
- [Привязка к документу](#)
- [Области видимости](#)

Формальный синтаксис описание типа счетов параметров

```
$ОписаниеТипа      = [Расширение] ( "Тип" | "Type" )  
                    ИмяТипа [Предки] ";" "  
                    ( [{ОписаниеПеречисления}] |  
                    {Параметр} ) |  
                    [<РазделяющиеПараметры>]  
                    Конец ";" "  
  
$Расширение        = ( "Расширяет" | "Extends" )  
$ИмяТипа            = Идентификатор  
$Предки            = ( "Inherited" | "Унаследованный" )  
                    ИмяТипа [ {", " ИмяТипа} ]  
$Параметр          = ( "Параметр" | "Parameter" )  
                    ИмяПараметра  
                    ":" ТипПараметра  
                    [ЗначениеПоУмолчанию]  
                    [", " Модификатор]  
  
$ИмяПараметра      = Идентификатор  
$ТипПараметра      = (  
    ( [ "Измеритель" | "Unit" ]  
      ИмяСправочникаИзмерителей  
    )  
    | ИмяСправочника  
    | ( "Строка" | "String" )  
    | ( "Целое" | "Integer" )  
    | ( "Логическое" | "Logical" )  
    | ( "Число" | "Numeric" )  
    | ( "Дата" | "Date" )  
  )  
$ЗначениеПоУмолчанию = "= " ( [Число "^" ] ИмяЭлементаСправочника  
    | СтрокаВКавычках  
    | Целое  
    | ( "Истина" | "True" | "Ложь" | "False" )  
    | Число  
    | Дата  
    | "Nil"  
    | Формула  
    | Функция  
  )  
$Модификатор       = Обратный | Разделяющий  
                    | Индексируемый | Разбиение  
                    | Аккумулируемость | [", " Модификатор]  
$Оборотный         = "Оборотный" | "Turn"
```



```

$Разделяющий      = "Разделяющий" | "Separator"
$Индексируемый    = "Индексируемый" | "Indexed"
$Разбиение        = "Неразбиваемый" | "Unsplitted"
$Аккумулируемость = "Накапливается" | "Accumulated"
                  | "Ненакапливается" | "NotAccumulated"
$Название         = ( "Название" | "Title" ) СтрокаВКавычках
$РазделяющиеПараметры = Разделяющий ИмяПараметра { ',' ИмяПараметра } ';' ;
$Разделяющий      = "Разделяющий" | "Separator";

```

Описание типа счета. Модификатор Разделяющий|Separator

Стандартное описание типа счетов начинается с ключевого слова **Тип|Type**, за которым указывается имя типа счета, далее следует перечень параметров типов счета и заканчивается описание ключевым словом **End|Конец**, например:

```

тип Базовый;
  Visible
    параметр Сумма :измеритель спрЕдИзм = 0.00^руб;
  ...
конец;

тип ТРасчеты inherited Базовый;
  параметр Контрагент      :спрКонтрагент;
  параметр Сделка          :спрСделка;
  параметр СчФактураВх     :спрСчФактураВх, оборотный;
  параметр СчФактураИсх    :спрСчФактураИсх;
  Separator Контрагент, Сделка;
конец;

```

Параметры счетов могут иметь один или несколько [модификаторов](#). По умолчанию для каждого параметра счета подсчитываются как остатки, так и обороты. Если по какому-либо параметру нужно получить свернутого сальдо, то в описание параметра добавляется модификатор [Оборотный](#). Параметры конкретного типа счетов, по которым необходимо получить разделенное сальдо (т.е. фактически иметь два показателя для одного параметра - по дебету и кредиту), в описании перечисляются после модификатора **Разделяющий|Separator**.

Модификатор **Разделяющий** не является атрибутом параметра типа счета, а является свойством типа счета и разрешает получить разделенное сальдо для параметров Контрагент, Сделка по всем счетам типа ТРасчеты. Однако, аналитические параметры Контрагент и Сделка, *принадлежащие другим типам счетов*, могут иметь разделенное сальдо.

Модификатор **Разделяющий** можно использовать также и в расширениях типов счетов.

Описание параметров типов счетов

Каждый параметр характеризуется своим типом данных. В качестве типов данных могут использоваться стандартные типы (Строка|String, Целое|Integer, Число|Numeric, Дата|Date, Логическое|Logical), а также [перечислимые типы](#).

Помимо указанных типов формальные параметры могут иметь тип **Справочник**. Если параметр объявлен как:

- **измеритель** - в описании параметра указывается ключевое слово **измеритель** и имя справочника единиц измерения или валют;
- **аналитический параметр** - в описании параметра указывается имя справочника [общего назначения](#).

Например:

```

тип Базовый;
  параметр Сумма :измеритель спрВалюта = nil;
конец;

```

В этом случае в проводках и полупроводках указываются:

- **два значения** (числовое значение и единица измерения) - для справочников единиц измерения и валют;
- **одно значение** (числовое) - для справочников общего назначения.

Параметры типов счетов могут иметь [псевдонимы](#).

Значения параметров по умолчанию

Допускается описание значения параметра по умолчанию, что позволяет в диалогах ввода проводок и операций подставить рекомендуемое разумное значение, а в типовых операциях и текстовых журналах

несколько сократить код. Значения по умолчанию можно задавать для всех параметров, кроме строковых (не имеет смысла) и логических (они и так не хранятся). В качестве значения по умолчанию может быть указано **nil**, что обозначает пустое значение параметра. Для аналитических параметров это значение по умолчанию. Для измерителей это обозначает нулевую сумму или нулевое количество и единицу измерения по умолчанию.

Значения по умолчанию в описании параметров счетов указываются после знака "=" и могут содержать простые выражения со стандартными типами и операциями языка ТБ.Скрипт, а также вызов функций. Пример:

```
тип ТипСчета;  
    параметр Сумма : измеритель спрВалюта = 1000^руб;  
    параметр Сумма2 : измеритель спрВалюта = Сумма+1^руб;  
    параметр Цена : число = GetVariable("ОбычнаяЦена");  
конец;
```

В приведенном выше примере, если в проводках не будет задана сумма, то по умолчанию будет подставляться значение 1000.

Для уменьшения памяти, отводимой под проводку на сервере расчетов, можно для каждого параметра проводки использовать ключевое слово **Default**, указанное после него *значение не будет храниться в проводках*. В качестве такого значения следует указывать наиболее часто употребляемые в проводках значения.

Если слово **Default** не задано, то по умолчанию в проводке не будут храниться нулевые значения nil (0 - для простых типов, единица измерения - для измерителей).

Пример:

```
параметр Сумма : измеритель Валюта, default Руб; -- измеритель  
параметр ТипПлатежа : ТипПлатежа = Налоги, default Налоги; -- перечисление  
параметр ТипДокумента : Integer = 0, accumulated, default 1; -- целое
```

Наследование типа счета

Наследование типа, задаваемое с помощью ключевого слова **Унаследованный|Inherited**, означает, что новый тип будет содержать все параметры родительского типа, а также все те, что добавлены непосредственно в описание нового типа.

```
тип Базовый;  
    параметр Сумма : измеритель спрВалюта = nil;  
конец;  
  
тип ОстатокТовара унаследованный Базовый;  
    параметр Количество : измеритель Единица;  
    параметр Товар : Товар;  
    параметр Склад : Склад = nil;  
конец;  
  
тип ДляИзделий унаследованный Базовый;  
    параметр КодПродукта : Целое = 0;  
    параметр ДатаВыпуска : Дата = 01.01.2000;  
    параметр Объем : Число = 1000.0;  
    параметр Сборка : Логическое = false;  
конец;
```

Расширение типа счета

Описание типа счета может быть расширено либо в этом же проекте в последующих фрагментах структуры учета, либо в других проектах, использующих текущий проект в качестве подпроекта. Синтаксис описания расширяемого типа счетов совпадает со стандартным описанием типа за исключением того, что перед ключевым словом **Тип|Type** идет слово **Расширяет|Extends**. Идентификатор типа в описании расширения должен совпадать с одним из определенных ранее типов счетов (в этом же проекте или [подпроектах](#)), например:

```
расширяет тип Базовый;  
    параметр СуммаВЕвро : измеритель Валюта = nil;  
конец;
```

В результате описания расширения существующий тип счета дополняется новыми полями. Новый тип счетов при этом не вводится, что отличает расширение от наследования. Тип счета не может быть описан одновременно наследником базового типа и расширяющим тот же или еще какой-либо тип. Иными словами,

модификаторы **Расширяет** и **Унаследованный** являются взаимоисключающими.

Привязка к документу

Для [новой модели журналов](#) (в описании используется свойство Version2|Версия2) может осуществляться привязка типа счета к документу или к подтаблице, если в привязке использовано ключевое слово **Struct**, чтобы можно было передавать параметры типовой операции, не модифицируя саму операцию. Имя документа записывается после ключевого слова **from**. При описании привязки параметров в типах счетов можно отдельно привязать дебет и кредит параметра, при этом привязка кредитовой части идет после дебетовой через пробел. Например:

```
тип УправленческийСчет from ОперДокумент Struct Позиции;  
  параметр Количество :измеритель sprЕдИзм = nil,  
    from "Позиции.КоличКуда-Позиции.ЕдИзмКуда"  
    "Позиции.КоличОткуда-Позиции.ЕдИзмОткуда";  
  параметр СерНомер :sprСерНомер = nil, некапливается,  
    from Позиции.СерНомерКуда Позиции.СерНомерОткуда;  
  ...  
  параметр Заказ :sprПроцесс = nil, некапливается, from Позиции.Заказ;  
  параметр ГТД :sprПроцесс = nil, некапливается, from Позиции.ГТД;  
  ...  
end;
```

Если параметр объявлен как измеритель, то привязка осуществляется по двум полям, для остальных параметров - по одному (Позиции.СерНомерКуда,Позиции.ГТД). Сначала указывается поле для числового значения (Позиции.КоличКуда), а затем после знака "-" записывается второе поле - для единицы измерения (Позиции.ЕдИзмКуда).

Области видимости

Параметры типов счетов могут описываться в видимой и невидимой областях, что определяется ключевыми словами **Visible|Видимо** и **Hidden|Скрыто**, например:

```
тип Базовый;  
...  
visible  
  параметр Сумма :измеритель sprВалюта = nil;  
  параметр Комментарий synonym Comment :String = "";  
конец;
```

Если не указана никакая область, то по умолчанию используется область **Visible**, в которой все параметры являются *видимыми параметрами*. Области могут чередоваться в произвольном порядке, допускается иметь несколько одинаковых областей в одном типе счета. Параметры, указанные в области **Hidden**, не отображаются в стандартных и прикладных средствах отображения и ввода проводок и параметров, а в [окне структуры учета](#) слева от таких параметрах будет нарисован "замочек".

При наследовании типов счетов, а также при описании расширения типа (Extends), допускается перенос параметра из одной области в другую с помощью синтаксиса переопределения параметров:

```
parameter P1;  
Где P1 - ранее описанный параметр.
```

Такой синтаксис можно использовать для описания в базовом типе счета невидимых по умолчанию параметров, а в типах счетов - наследниках делать их видимыми.

В процессе работы с программой объем хранимой в базе данных информации увеличивается, поэтому резко возрастает время обработки данных. Для уменьшения количества обрабатываемых документов в программе предусмотрена операция "закрытия" периода, которая запрещает редактировать документы и проводки задним числом, что особенно актуально после сдачи баланса.

Под *периодом закрытия* понимается период времени от даты закрытия предыдущего периода до даты закрытия текущего периода, после которой не допускается изменение данных в проводках и документах, сформированных позднее этой даты. Закрытие периода можно выполнять неоднократно с любой заданной периодичностью, которая является прерогативой самого пользователя.

Предупреждение. Закрытие ("выверка") периодов разрешено только для областей учета, имеющих модификатор **Closeable|Закрываемая**.

Операция закрытия периода выполняется в окне "Управление закрытыми периодами", которое открывается командой Закрытие периода. В левой части окна перечислены области учета. Если никаких операций, связанных с закрытием периода, не проводилось, то папки с областями учета пустые. В этот момент доступна только команда **Добавить уровень**, а команда **Закрыть период** недоступна.

Для проведения *первой операции закрытия периода* необходимо выделить нужную область учета и выполнить команду **Добавить уровень** и в диалоге указать дату закрытия периода. Каждому уровню присваивается свой номер. Разрешается изменять (увеличивать или уменьшать) дату закрытия периода. После установки даты закрытия становится доступна команда **Закрыть период**, позволяет увеличить последнюю дату закрытия периода.

Если последнюю дату закрытия требуется уменьшить, то следует удалить последнюю операцию закрытия периода командой **Удалить** и в диалоге "Удаление закрытия" не указывать дату начала периода, а после этого снова вызвать команду **Добавить период** и задать новую дату закрытия. При проведении операции закрытия номер уровня не изменяется.

Предупреждение. Удаление производится, начиная с последнего уровня, промежуточный уровень удалить нельзя.

Операцию закрытия периода можно выполнять со сверткой данных (сворачиванием данных по определенным критериям с целью уменьшения объема информационной базы)или без нее, тогда в диалоге следует снять флаг **Производить свертку**.

Все последующие вызовы команды **Добавить уровень**, кроме первого вызова, необходимы для добавления новых уровней с целью увеличения степени сжатия исходных данных. Причем если сжатия данных не требуется, то добавлять новые уровни не нужно. Чем выше уровень, тем сильнее свернуты данные, за счет многократного сворачивания данных. Достигается это за счет объединения нескольких проводок, имеющих одинаковую аналитику или одинаковые счета дебета и кредита, в одну в одну, так называемую сводную проводку. Критерии, по которым происходит объединение проводок, задаются пользователем за счет установки/снятия соответствующих флагов. На каждом следующем уровне данные будут более плотно упакованы.

В результате проведения каждой операции закрытия все данные сохраняются в виде записи в отдельной таблице, тем самым значительно сокращая размер базы данных. Для просмотра таблицы нужно открыть окно и в дереве записей выбрать класс Kernel и открыть папку Fixed, в которой находятся записи журнала для каждой области учета, попавшие под закрытие/свертку периода. В столбце **Уровень** указывается номер уровня.

Типовой операцией называется поименованная совокупность логически связанных [проводок](#) (полупроводок), описывающая стандартную хозяйственную операцию.

Типовые операции позволяют существенно упростить и ускорить ввод информации о наиболее распространенных хозяйственных операциях, а также унифицировать их представление в программе.

Типовые операции описываются на специальном языке типовых операций в файлах с расширением COD (в предыдущих версиях Турбо Бухгалтера использовалось расширение DEF). Каждая операция, по аналогии с классами ТБ.Скрипт и их методами, идентифицируется полностью квалифицированным именем, уникальным в контексте всего проекта. С помощью данного имени операция может быть вызвана из журналов хозяйственных операций или же из других типовых операций. При необходимости операция может иметь список параметров, влияющих на логику ее работы. Непосредственно тело операции составляют инструкции (например, проводки и вызовы встроенных функций), задающие учет и движение денежных и прочих средств по сущностям структуры учета.

В этом разделе подробно рассматривается язык типовых операций:

- [Язык типовых операций](#)
- [Типы данных](#)
- [Допустимые синтаксические умолчания](#)
- [Массивы](#)
- [Описание процедур, функций и операций](#)
- [Вызов процедур и функций](#)
- [Вызов типовых операций](#)
- [Формирование полупроводок и проводок. Дебет. Кредит](#)

С некоторыми ограничениями процедуры и функции могут быть вызваны из кода типовых операций и непосредственно из журналов.

Вызов функций возможен из файлов типовых операций и журналов, а вызов процедур - только из файлов типовых операций. Следует иметь в виду, что вызываться могут не только процедуры и функции, описанные на ЯТО, но и методы некоторых системных классов (например, **Система** и **Бухгалтерия**).

Формат вызова функций и процедур полностью соответствует текущим соглашениям для языка ТБ.Скрипт:

<

В ЯТО вызов функции может входить в состав выражения и, в частности, служить источником значения для переменной. При вызове функции из текстового журнала она должна обязательно входить в состав выражения.

При вызове процедур и функций используется позиционный способ передачи параметров, то есть фактические параметры перечисляются в том же порядке, в котором объявлены формальные параметры в описании метода. При несоответствии количества фактических и формальных параметров или их типа генерируется ошибка.

Язык типовых операций (ЯТО) допускает вызов других типовых операций из кода типовой операции. Кроме того, очевидно, что вызов типовых операций возможен и из журналов - именно для этого типовые операции в основном и предназначены. Форматы вызова операций из файлов типовых операций и из журналов различаются.

Из COD-файла операция вызывается следующим образом:

```
ИмяОперации [ИмяПараметра1=ЗначениеПараметра1  
[ ,ИмяПараметра2=ЗначениеПараметра2] [ , ... ] ];
```

При вызове из журнала перед именем операции ставится символ двоеточия (:), весь оператор должен быть записан целиком на одной строке, а завершающий символ точки с запятой (;) не допускается.

```
: ИмяОперации [ИмяПараметра=ЗначениеПараметра  
[ ,ИмяПараметра=ЗначениеПараметра] [ , ... ] ]
```

Таким образом, при вызове типовых операций используется поименованный способ передачи параметров. Порядок следования фактических параметров не имеет значения и может не совпадать с порядком объявления формальных параметров. Параметры, имеющие значение по умолчанию, могут быть опущены. Если опущен параметр, не имеющий значения по умолчанию, генерируется ошибка при компиляции.

Примеры вызовов типовой операции из COD-файла:

```
ВводОстатка Сумма=100^usd, ИнвНом=Станок1, Норма_Ам=10, Износ=1.5;  
ВводОстатка Сумма=500.99^руб, ИнвНом=Станок2, Комментарий="Станок2";
```

То же самое в текстовом журнале выглядит следующим образом:

```
: ВводОстатка Сумма=100^usd, ИнвНом=Станок1, Норма_Ам=10, Износ=1.5  
: ВводОстатка Сумма=500.99^руб, ИнвНом=Станок2, Комментарий="Станок2"
```

Особый случай представляет вызов типовых операций из картотечных журналов. Здесь вызов осуществляется автоматически на основе привязки типовой операции к полям картотечного журнала с помощью редактора проекта. При этом имя типовой операции, а также фактические параметры для аргументов операции указываются в [описании свойств журнала](#).

Вставка и редактирование операций в табличном журнале осуществляется с помощью группы соответствующих [команд и диалогов](#).

В ЯТО разрешается опускать имя плана счетов, если он является планом по умолчанию. Например, если план "Основной" описан в структуре учета с модификатором БАЛАНСОВЫЙ, то синтаксически правильной будет следующая упрощенная запись:

Другое правило умолчания - это разрешение опускать имя аналитического справочника или плана счетов при наличии контекста, который позволяет это имя однозначно подразумевать. Например, в вышеприведенном примере второй параметр процедуры П1 имеет тип **Unit Валюты**, поэтому при ее вызове достаточно написать "10^руб", вместо "10^Валюты.руб". Так как подобный контекст всегда существует, то всегда можно опускать имя справочника.

Так же как и в языке ТБ.Скрипт, в ЯТО возможно использование [массивов](#). Синтаксис и семантика совпадают с применяемыми в ТБ.Скрипт. Например:

;

При написании типовых операций бывает удобно объединять логически связанные участки кода в процедуры и функции.

Процедуры и функции могут вызываться из ЯТО или из журналов хозяйственных операций. Процедуры и функции не видны в иерархии типовых операций, доступной пользователю из интерфейса программы. Кроме того, их нельзя ввести в журналы с помощью соответствующего диалога ввода.

Синтаксис описания процедур и функций такой же, как и в языке [ТБ.Скрипт](#).

func |

Операции - это процедуры особого рода, они имеют расширенный синтаксис, позволяющий описывать комментарии к самой операции и к ее параметрам.

Операции формируют иерархию типовых операций, доступную из стандартного интерфейса программы, и могут быть введены в журнал с помощью диалогов ввода. Другими словами, операции - это интерфейсные сущности, видимые пользователю, тогда как процедуры и функции - внутренние, служебные сущности.

Синтаксис:

Синтаксис описания операции включает следующие конструкции языка, записанные в заданном порядке:

- ключевое слово **Опер/Oper**;
- идентификатор операции, при необходимости дополненный комментарием;
- необязательный список параметров, заключенный в круглые скобки;
- тело операции, состоящее из операторов языка типовых операций (ЯТО), подобный языку ТБ.Скрипт, но со специфическими расширениями и ограничениями;
- ключевое слово **Конец/End**, за которым следует точка с запятой.

Список параметров представляет собой последовательность описаний параметров, разделенных символом ';' (точка с запятой). Описание каждого параметра начинается с его идентификатора, за которым указывается необязательный комментарий, а затем через двоеточие - тип. После имени типа при необходимости указывается значение по умолчанию.

Пример типовой операции:

При описании типовых операций доступны все [простые типы данных](#), определенные в ТБ.Скрипт. Семантика использования этих типов полностью идентична ТБ.Скрипт. Дополнительно к простым типам данных в языке типовых операций вводятся следующие типы:

- [Счет | Account;](#)
- [Признак | Sign;](#)
- [Измеритель | Unit.](#)

Тип Счет|Account

Значение типа **Счет** - идентификатор конкретного счета из [структуры учета](#). Так как каждый счет входит в тот или иной план счетов, полностью квалифицированное значение типа **Счет** имеет формат:

<ИмяПлана>.<ИмяСчета>

Например:

```
Var x : Счет;  
x = Баланс.01;
```

Тип **Счет** является абстрактным базовым типом для всех типов счетов, описанных в структуре учета. В ЯТО возможно описание переменных любого конкретного типа счетов из числа тех, что указаны в структуре учета. Так как типы счетов, в противоположность счетам, не относятся к конкретному плану счетов, для формирования идентификатора типа счета используется упрощенный синтаксис - просто имя счета:

<ИмяТипаСчета>

Например:

```
var Сч1 : Базовый;
```

при условии, что в структуре учета есть примерно следующее описание:

```
тип Базовый;  
  параметр Сумма :Валюта = nil;  
  параметр Комментарий :String = "";  
конец;
```

Тип Признак|Sign

Значение типа **Признак** является идентификатором элемента аналитического справочника, описанного в структуре учета. С точки зрения объектно-ориентированного подхода, используемого в Студии, аналитический справочник - это класс, а элемент аналитического справочника - экземпляр этого класса (объект). Таким образом, переменная типа **Признак** - это объект специального класса. Для предопределенных элементов аналитических справочников (они задаются непосредственно в файле [структуры учета](#)) считается, что соответствующие объекты существуют всегда, поэтому возможно задание констант типа **Признак**. Его формат следующий:

<ИмяСправочника>.<ИмяЭлемента>

Например:

```
Валюты.руб
```

Стоит еще раз обратить внимание, что описание констант типа **Признак** возможно только для предопределенной (обязательной) аналитики.

Тип **Признак** является абстрактным базовым типом для всех аналитических справочников, описанных в структуре учета.

Для задания переменной, хранящей значения из конкретного справочника, необходимо в качестве идентификатора типа использовать имя данного справочника, которое должно быть описано в структуре учета. Например, для справочника "Валюты" это выглядит так:

```
var Вал :Валюты;
```

При описании переменных и параметров рекомендуется использовать конкретные типы счетов и справочники. Это вызвано тем, что на этапе компиляции для переменных, описанных просто как **Счет** или **Признак**, неизвестен состав свойств, а значит при работе с ними (чтении свойств, генерировании проводок и т.п.) необходимо использовать [динамическую диспетчеризацию](#), что существенно влияет на скорость обработки данных.

Тип Измеритель|Unit

Значение типа **Измеритель** представляет собой составную конструкцию, в которую входит число и ссылка на элемент справочника, содержащего измерители.

Для описания идентификатора типа **Измеритель** используется синтаксис:

```
Unit <Analytic Identifier>
Измеритель <Идентификатор справочника>
```

Например:

```
func Measure (Sum :Real; Ref :Валюты) :Unit Валюты;
    return Sum^Ref;
end;
```

Как видно из примера, для конструирования значения типа **Измеритель** используется оператор ^. Левый операнд этого оператора - вещественное или целое число, а правый - элемент аналитического справочника. Результат применения оператора имеет тип конкретного измерителя. Этот оператор имеет такой же приоритет, как и операции * (умножения), / (деления), % (получения процента от числа).

Возможно описание переменных-измерителей и параметров-измерителей с абстрактным признаком, для чего в описании вместо имени справочника указывается базовый класс **Признак | Sign**:

```
Unit Sign
Измеритель Признак
```

Допускается сокращенная запись, в которой ключевое слово **Sign / Признак** опускается. То есть тип:

```
Unit
Измеритель
```

эквивалентен любому из двух вышеприведенных.

Для работы с переменными и выражениями типа **Unit Sign** (или **Unit <Конкретный признак>**) в ТБ.Скрипт имеется две функции класса **Бухгалтерия**:

[ЗначениеИзмерителя / UnitValue](#)
[ПоказательИзмерителя / UnitFactor](#)

Первая из них возвращает числовую часть измерителя, а вторая - признак (идентификатор) единицы измерения.

Над измерителями разрешено выполнение всех основных операций логического сравнения: "=", "<>", ">", "<", ">=", "<=". Первые две из них допускается выполнять над любыми парами измерителей, а остальные - только над теми измерителями, показатели которых совпадают. Например, можно сравнить "на больше/меньше" две суммы в рублях, но нельзя сумму в рублях - с суммой в долларах. Хотя доллары и рубли, как правило, связаны в проекте между собой курсом, в контексте операции сравнения не известна дата, для которой следовало бы брать курс. Тем более очевидно, что нельзя сравнить вес в килограммах с емкостью в литрах.

Операции сравнения измерителей эквиваленты следующему коду:

```
M1, M2 :Unit Sign;
M1 > M2 ~ UnitValue(M1) > UnitValue(M2)
M1 < M2 ~ UnitValue(M1) < UnitValue(M2)
M1 >= M2 ~ UnitValue(M1) >= UnitValue(M2)
M1 <= M2 ~ UnitValue(M1) <= UnitValue(M2)
```

Для родственных по показателям измерителей определены также и арифметические операции сложения, вычитания, умножения на число, деления на число, получения процентной части измерителя:

```
M1, M2 :Unit Sign;
M :Unit Sign;
F :Numeric;

M1 + M2 ~ (UnitValue(M1) + UnitValue(M2))^UnitFactor(M1)
M1 - M2 ~ (UnitValue(M1) - UnitValue(M2))^UnitFactor(M1)

M * F ~ (UnitValue(M)*F) ^ UnitFactor(M)
M / F ~ (UnitValue(M)/F) ^ UnitFactor(M)
M % F ~ (UnitValue(M)%F) ^ UnitFactor(M)
```

Пример:

```
-- перевод суммы в любой валюте в сумму с показателем "To"
func ConvertEx (From :unit; To :Sign; Dt :Date) :unit;
    return Convert(UnitValue(From), UnitFactor(From), To, Dt)^To;
end;

proc ButStartClick(Sender :String);
    var Q :HTransQuery;
```

```

var H :HTrans;
var I :Integer;
var locSum :unit Валюта;

-- запрос всех полупроводок
Q = HTransQuery.Create("");

ClearTrace;
Trace(Str(class HTransQuery(Q.ClassType).ClassName));

I = 0;
locSum = 0^руб;
while not Q.EOF do
  H =Q.Current;
  Trace( Str(I) + ", " + Str(H.Date) + ", " +
    Str(H.Счет) + ", " + Str(H.Сумма)+
    Str(H.Document) );
  locSum = locSum + ConvertEx(H.Сумма, UnitFactor(locSum), Today);
  I = I + 1;
  Q.Next;
end;
Trace('Итого:' + Str(locSum));
Trace('20%:' + Str(locSum % 20));
end;

```

Для трех вышерассмотренных типов **Счет**, **Признак** и **Измеритель**, специфических для бухгалтерского ядра, могут применяться общие правила [приведения типов](#). Например, величина типа **Измеритель** может быть записана в переменную типа **Вариант**, и наоборот, сохраненный в переменной типа **Вариант** измеритель может быть преобразован к исходному типу измерителя. При этом на стадии выполнения кода программой выполняется проверка того, что приведение корректно (так, нельзя **Измеритель Упаковки** привести к **Измеритель Валюты**).

```

proc P1(V :Variant); var Money: Unit Валюта;
var B: Unit Sign;
-- приводим Вариант к измерителю
B = Unit Sign (V);
-- для любого измерителя выводим:
trace(UnitValue(B)); -- число
trace(UnitFactor(B)); -- и единицу измерения
-- а для денежных сумм дополнительно - курс
if V is Unit Валюта then
  -- приводим Вариант к денежному измерителю
  Money = Unit Валюта (V);
  trace(UnitFactor(Money).Rate[Today]);
end;
end;

proc P2;
  P1(100^Валюта.Руб)
  P1(5^Упаковка.Ящик);
end;

```

В языке типовых операций (ЯТО) существует три встроенных специализированных процедуры для формирования проводок и полупроводок: Дебет(Debet), Кредит(Credit), Проводка(Transaction). Именно эти процедуры оперируют учетными данными в машине проводок.

Данные процедуры в силу своей специфики и исключительности не подчиняются обычным правилам использования процедур в описаниях типовых операций. Перечислим их отличительные особенности:

- Вызов этих процедур допустим как из файлов типовых операций, так и из журналов (см. [концепцию полупроводок и проводок](#));
- При вызове используется комбинированный вариант передачи параметров, включающий как поименованный, так и позиционный способы;
- Имена формальных параметров не фиксированы и зависят либо от первого (в случае процедур Дебет и Кредит), либо от двух первых (в случае процедуры Проводка) фактических параметров.

Формат вызова процедур **Дебет/Кредит** (полупроводок) следующий:

Первый обязательный параметр процедур Дебет/Кредит имеет тип **Счет**, и его фактическое значение записывается непосредственно после имени процедуры, то есть оно не предваряется именем формального параметра, как все остальные. Таким образом, получается комбинированный - "позиционно/поименованный" способ передачи параметров.

Остальные параметры передаются стандартным поименованным способом. Имена формальных параметров процедур Дебет/Кредит должны совпадать с именами параметров счета, описанных в структуре учета для данного типа счета.

Параметры, которые имеют значение по умолчанию, допускается опускать. В таких случаях в качестве значения параметра используется значение по умолчанию. Остальные параметры должны быть обязательно присвоены.

Пример:

Формат вызова процедуры **Проводка**:

По выполняемому действию проводка заменяет пару процедур Дебет/Кредит, сохраняя понятие корреспонденции между счетами. Счета записываются двумя первыми обязательными параметрами.

Множество остальных параметров представляет собой объединение множеств параметров обоих счетов. Перед именем формального параметра может идти символ "+" (плюс) или "-" (минус) - это является явным указанием того, к какому счету (соответственно, дебета или кредита) относится данный параметр. При этом соблюдается следующее правило: если оба счета имеют одноименные параметры и их типы совпадают, то указание в проводке значения этого параметра без символов "+/-", присваивает значение обоим параметрам.

Пример:

Ниже приведен пример описания типовой операции, вызывающей процедуру **Проводка**.

`transaction`

`transaction`

При формировании проводок из типовых операций можно воспользоваться функцией [CurrentDomain](#), если требуется узнать, в контексте какой [области учета](#) вызвана текущая типовая операция. Функция возвращает имя текущей области учета.

Использование областей учета, описываемых в [структуре учета](#), позволяет разделить весь объем учетных данных на несколько независимых наборов, в соответствии с особенностями, например, бухгалтерский и налоговый учет. Это положительно сказывается на быстродействии программы.

Благодаря функции **CurrentDomain**, в тексте типовой операции можно осуществить избирательную обработку учетных данных и генерацию проводок в зависимости от целевой области учета.

Синтаксис и семантика языка типовых операций (ЯТО) практически ничем не отличается от синтаксиса и семантики [языка ТБ.Скрипт](#), что существенно облегчает освоение и применение инструментария. По сути, ЯТО можно представить, как язык ТБ.Скрипт, в котором разрешено использование лишь ограниченного круга встроенных классов - **АвтоОбъект**, **ТекстовыйФайл**, **Система**, **Бухгалтерия**, **ЗапросПолупроводок**. В то же время в ЯТО определены [дополнительные типы данных](#) (такие как **счет**, **измеритель** и **признак** - элемент аналитического справочника) и специальные языковые конструкции (например, [проводки и полупроводки](#)). Иными словами, в ЯТО запрещено использование визуальных и интерфейсных объектов, взаимодействующих с пользователем, а также объектов для работы с информационной базой. Вызвано это особенностями использования ЯТО и непосредственно его назначением.

Структура COD-файла с описанием типовых операций аналогична структуре COD-файла с описанием класса ТБ.Скрипт. Начинается описание с заголовка, в котором используется ключевое слово [Класс / Class](#) и, при необходимости, в кавычках приводится развернутое название группы типовых операций, содержащихся в данном файле. Фактически, каждая группа типовых операций образует в иерархии проекта класс, унаследованный от абстрактного класса [Операции](#). Сами типовые операции выступают в качестве методов (процедур и функций) этого класса. Их программирование подчиняется правилам, общим для классов языка [ТБ.Скрипт](#).

Точно так же? как и в обычных COD-файлах, в файлах типовых операций с помощью ключевых слов [public/публично](#) и [private/лично](#) могут быть явно заданы фрагменты с общедоступными и/или внутренними методами.

В отличие от пользовательских классов, бланков и картотек, в типовых операциях нельзя использовать директиву **InObject**. Все свойства и методы класса типовой операции считаются определенными в контексте класса (т.е. подразумевается, что после заголовка класса неявно указана директива **InClass**). Вызвано это тем, что большинство методов такого класса, исходя из его назначения, реализуют конкретные типовые операции и, следовательно, должны быть доступны для машины проводок в любой момент выполнения проекта. В то же время, свойства и методы объектов требовали бы предварительной инициализации экземпляров объектов.

Также в типовых операциях следует осмотрительно использовать глобальные переменные (как собственно классов типовых операций, так и внешних классов). Если глобальная переменная изменяется в теле типовой операции, это будет происходить при каждом вызове данной операции (при обработке соответствующих журналов). Поскольку обработка журналов может выполняться частично и непредсказуемо зависит от последовательности действий пользователя, результат изменения глобальной переменной становится случайным. Как правило, глобальные переменные используются в типовых операциях лишь как источники значений (например, ставок налогов).

В методах типовых операций допускается использовать все [операторы ТБ.Скрипт](#), в том числе:

- **если / if;**
- цикл **пока / while;**
- цикл **для / for.**

Заканчивается описание группы операций ключевым словом **конец/end**.

Одно из основных отличий типовых операций от прикладных классов, реализованных на языке ТБ.Скрипт, заключается в том, что операции выполняются не клиентской частью Студии, а серверным модулем (это верно даже в том случае, если прикладной проект установлен на автономном компьютере). В связи с этим, отладка исходного кода типовой операции возможна лишь в ограниченном виде - встроенный [отладчик](#) может использоваться только при соответствующих настройках, т.е. при включении отладки на сервере расчетов, что автоматически исключает возможность отладки кода на языке ТБ.Скрипт на клиентском компьютере.

Каждая запись (документ), используемая в программе, имеет заранее известную *модель (или структуру)*, т.е. набор параметров, значения которых, собственно, и отличают один документ от другого. Документы разного типа, как правило, имеют различную структуру данных, а однотипные документы имеют одинаковый набор реквизитов. Например, практически в любом бухгалтерском документе содержатся наименование предприятия, дата составления, подписи ответственных лиц.

Проекты строятся на принципах объектно-ориентированного программирования, в терминах которого тип документа называется классом, причем описание такого класса как раз и задается структурой данных.

Для того чтобы сформулировать и описать структуру данных, а иными словами - определить классы документов, разработан специальный язык MTL. Его синтаксис и правила рассматриваются в темах:

[Состав файла описания структуры базы данных](#)

[Заголовок MTL-описания](#)

[Описание пользовательских типов](#)

[Описание документов \(записей\)](#)

[Расширенный синтаксис MTL](#)

[Реорганизация базы данных](#)

Описанные в MTL-файлах классы документов отображаются (после компиляции) в [иерархии объектов проекта](#) в разделе "Записи". Редактируются MTL-файлы с помощью встроенного [текстового редактора](#).

На базе описаний классов записей (документов) в прикладном проекте могут быть созданы [картотеки](#) - визуальные формы для просмотра и редактирования наборов документов, а также [бланки](#) - экранные формы для просмотра и редактирования конкретных экземпляров документов. Кроме того, записи (документы) отдельного класса могут использоваться для автоматического формирования [журналов хозяйственных операций](#).

Таким образом, язык описания модели данных позволяет создать информационную основу для большинства других составных частей проекта.

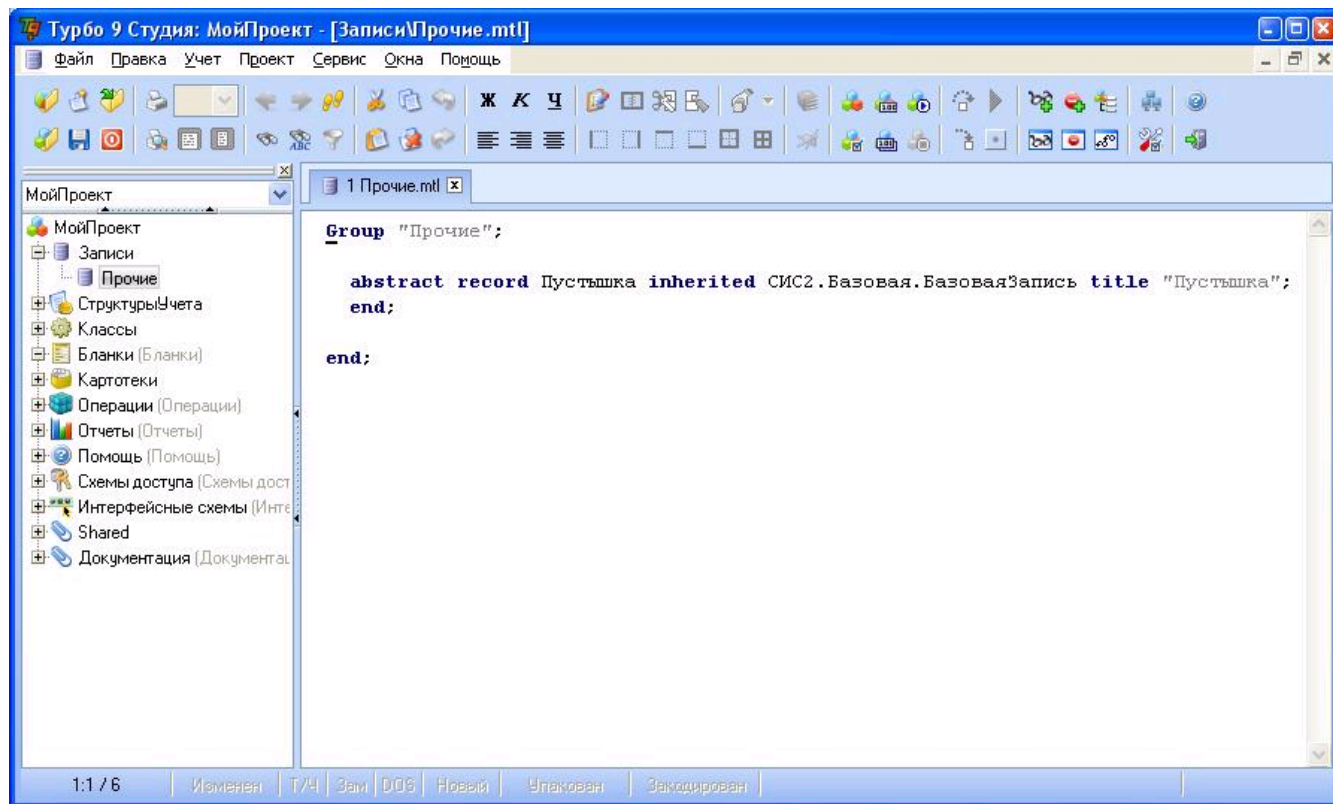


Рис. Язык описания модели данных MTL.

Описание документа так же, как и описание структуры данных в целом имеет составную структуру.

Внимание. В терминах программы документ называется записью, поэтому в Справочной системе вместо термина документ используется термин запись.

Описание любой записи состоит из блока **Report|Document** с заголовком, в котором задаются основные характеристики записи (название, признак иерархичности и т.д.), и непосредственно тела блока, где перечисляются поля, индексы, внешние ключи, подчиненные таблицы и другие параметры записи.

Синтаксис языка MTL, используемого для описания отдельной записи, приведен в темах:

[Заголовок записи](#)

[Описание полей записи](#)

[Описание многозначных полей](#)

[Служебные поля](#)

[Описание структур](#)

[Список уникальных ключей](#)

[Описание и назначение индексов](#)

[Описательные поля](#)

[Пользовательские ограничения](#)

[Внешний ключ](#)

Прежде чем перейти к изложению синтаксиса MTL-описания записей, сделаем небольшое теоретическое отступление, без которого было бы трудно понять суть MTL-описаний.

В контексте баз данных каждое MTL-описание записи формирует так называемую таблицу (или даже не одну, а несколько). Таблица представляет собой некую низкоуровневую структуру (ее устройство может изменяться от одной СУБД к другой и здесь не рассматривается), где, собственно, и хранятся экземпляры записей конкретного типа. На практике довольно часто возникает необходимость работать с записями, которые в силу своей специфики должны храниться не в одной таблице, а в нескольких. Например, запись (документ) "Накладная" имеет табличную часть с перечнем товаров. Данный перечень не может храниться в той же таблице (таблице базы данных), что и информация с реквизитами накладной - это обусловлено внутренней архитектурой современных реляционных СУБД. Поэтому на самом деле накладная разбивается на 2 части, каждая из которых отнесена к своей таблице. Каждый раз, когда накладная вызывается на экран, программа собирает всю информацию о ней из разных таблиц в единое целое и предоставляет пользователю. Обратный процесс - расщепление информации и разнесение по разным таблицам - происходит при сохранении записи. При этом одна таблица называется основной, а вторая подчиненной. Подчиненных таблиц может быть много или ни одной, в то время как основная таблица есть всегда, и она - только одна.

Таким образом, при описании записей в MTL-файле помимо простых или многозначных полей могут встречаться и ссылки на подчиненные таблицы, причем подчиненные таблицы не всегда описываются явно. В некоторых случаях программа позволяет выполнять действия по неявному формированию структуры данных, основываясь на внутренних умолчаниях, правила их применения рассматриваются особо. Так, [многозначные поля](#) фактически реализуются с помощью подчиненных таблиц, но прикладному программисту не обязательно описывать подтаблицы в MTL-файле явно - достаточно лишь описать многозначное поле, и в базе данных будет сформирована подчиненная таблица.

Отношение между основной и подчиненной таблицами могут иметь вид "один к одному" или "один ко многим". В описании записи вид отношения не задается - он определяется лишь в момент ее использования. Если таблица используется в однозначном поле, отношение "1:1", если в многозначном, то - "1:N".

Все идентификаторы, используемые в языке MTL (т.е. названия таблиц, полей и т.д.), могут записываться как по общим правилам идентификаторов, так и заключаться в кавычки. Например, запись имени поля *Номер* и "Номер" эквивалентны. При последующем описании синтаксиса кавычки не будут нигде применяться. Они необходимы только в тех случаях, когда при описании внешних групп записей (считываемых из баз данных сторонних программ) встречаются названия структур данных, не отвечающие правилам составления идентификаторов. Например, название таблицы "Сотрудник@Отдел" должно заключаться в кавычки во избежание ошибок компиляции MTL-файла.

Внешний ключ

Внешний ключ служит для идентификации записей при проведении операций импорта или экспорта данных. В качестве внешнего ключа может выступать служебное поле [ExtID](#). Однако, язык MTL, предназначенный для описания структуры данных разнообразных записей, позволяет для этих целей применять также *простое поле записи*.

Внимание. Только одно поле записи может быть объявлено внешним ключом.

Для описания внешнего ключа записи в языке MTL предусмотрено ключевое слово **ForeignKey**, а также его синонимы **ВнешнийКлюч**|**FKey**|**ВКлюч**. Для того чтобы простое поле записи можно было использовать как внешний ключ, его имя следует записать после ключевого слова **ForeignKey**, а в конце поставить символ ";".

Например:

```
Record ЕдИзм inherited ИерархическийСправочныйДокумент Title "Единица измерения";
  Field Название      :String(30);
  Field Описание      :String;
  Field Точность      :Integer;
  Field Базовая       :ЕдИзм;
  Field Умножать      :Boolean;
  Field Курс          :Numeric;  -- отношение к базовой ед. изм.

  ForeignKey Название;
end;
```

Значения полей, объявленных как **ForeignKey** используются в различных случаях, когда запрашивается описание у записи, не содержащей [описательные поля](#).

В случае необходимости при экспорте записей можно игнорировать имеющиеся в MTL-описании записей поля ForeignKey, используя пользовательский или программный интерфейс. Для этого в диалоге ["Экспорт ..."](#) на этапе режима экспорта полей имеется флаг **Игнорировать вторичный внешний ключ**, а в класс Экспортер - свойство [IgnoreForeignKey](#).

Для описания заголовка записи (документа) используется следующий синтаксис:

```
[Hierarchical][Abstract] Document <ИмяДокумента>
  [Inherited <ИмяРодительскогоДокумента>]
  [= <ИмяОсновнойТаблицы> [ ( <ИмяПоля> [:<ТипПоля>] ) ]]
  [Extends [[<ИмяПодпроекта>.]<ИмяГруппы>.]<ИмяРасширяемогоДокумента>]
  [Title "<Название>"]
```

Допускается использование следующих русскоязычных синонимов:

Ключевое слово	Синонимы
Document	Record, Запись, Документ
Hierarchical	Иерархический, Иерархическая
Extends	Расширяет
Abstract	Абстрактный, Абстрактная
Title	Название

Обязательная часть заголовка включает ключевое слово **Document** (или его синоним), после которого указывается название документа (без кавычек). Пример простого заголовка:

```
Document Накладная;
```

Все остальные модификаторы добавляются в заголовок по необходимости. Их назначение описывается далее в порядке частоты применения.

Важно, что названия документов в одной группе (MTL-файле) должны быть уникальными, хотя и могут совпадать в разных группах. Ссылки на документы внутри одной группы - "короткие" (только имя документа); ссылки на документы в другой группе - "полные" (ИмяБазы.ИмяДокумента).

Модификатор **Hierarchical** вводит важное свойство, обеспечивающее возможность группировать документы данного типа и организовывать их в иерархические структуры. Иными словами, если модификатор указан, то картотека документов может содержать группы неограниченной вложенности, а пользователь, после соответствующих настроек интерфейса программы, может создавать группы, перемещать и копировать документы между группами. Обращаем ваше внимание на тот факт, что в результате добавления свойства иерархичности все документы данного типа получают дополнительные [служебные поля](#) (**IsGroup** и **GroupDoc**), которые и обеспечивают построение иерархии.

Атрибут **Title** позволяет задать расширенное название (комментарий) документа. Фактически документы идентифицируются внутри проекта по имени, которое указывается после ключевого слова **Document**, а название (Title) используется только в справочных целях - в частности, при выводе сообщений пользователю системы.

Например, для картотеки юридических лиц **ЮрЛицо** наименованием может служить строка "Справочник юридических лиц". Название может быть не задано.

Наследование и классы документов

Рассмотрим использование атрибутов более сложного характера. Документы, как и большинство остальных сущностей Студии, имеет объектно-ориентированную природу, то есть представляют собой объекты определенных классов (типов). Фактически, это означает, что каждое описание документа в MTL-файле определяет новый класс.

В соответствии с базовыми принципами ООП, классы документов способны наследовать свойства других "документных" классов, что дает возможность упростить и унифицировать разработку прикладных систем. Например, если программист однажды описал класс документов "Операция" с основными реквизитами двух сторон торговой операции, то затем на основе этого класса можно определить производные классы первичных документов (например, "Счет", "Накладная"), добавив в них недостающие поля данных. Иными словами, производные документы описываются не "с нуля", а с учетом уже имеющейся структуры данных.

Наследование документов может осуществляться двумя принципиально разными способами: на уровне классов (наследование описания) и на уровне объектов (наследование данных). В первом случае используется ключевое слово **Inherited**, а во втором - **Extends**.

При наследовании на уровне класса описание документа-потомка неявным образом расширяется полями документа-родителя. Например, имеется описание базового класса документов:

```
Document Операция;
  Field Мы: Строка;
```

```
Field Контрагент: Строка;  
End;
```

Здесь определен класс документов "Операция" с двумя полями типа строка. [Описание полей документа](#) начинается с ключевого слова **Field**. Создадим производный класс:

```
Document ФинансоваяОперация Inherited Операция;  
Field Сумма: Число;  
End;
```

Производный класс "ФинансоваяОперация" имеет те же поля, что и базовый класс (несмотря на их отсутствие в описании), и кроме того дополняется полем **Сумма** числового типа. Фактически эта запись эквивалентна следующей:

```
Document ФинансоваяОперация;  
Field Мы: Строка;  
Field Контрагент: Строка;  
Field Сумма: Число;  
End;
```

Унаследованный документ получает все свойства своего родителя - полный набор полей, признак иерархичности, списки [уникальности](#) и [индексы](#).

На основе описания документов "Операция" и "ФинансоваяОперация" будут сгенерированы две независимые таблицы, хранящие экземпляры соответственно простых операций и финансовых. Важно отметить, что в производный документ переносится (в неявном виде) именно описание двух полей **базового документа**, а конкретные экземпляры документов этих двух классов будут содержать совершенно независимые данные (то есть в таблице документов "Операция" будут свои поля Мы и Контрагент, а в таблице документов "ФинансоваяОперация" - свои).

Второй вид наследования - **наследование данных** - по сути дела означает расширение существующей структуры данных (причем, вместе с хранящейся в таблице информацией!) новыми полями. Предположим, что класс документов "Операция" был описан в прикладном проекте "ОсновыУчета". Затем на основе данного проекта создается специализированный проект "Финансы", в котором требуется оперировать финансовыми операциями, использующими основные данные учета (из проекта "ОсновыУчета"). В терминах Студии проект "ОсновыУчета" становится [подпроектом](#) проекта "Финансы".

В этом случае логично расширить имеющийся в подпроекте "ОсновыУчета" класс "Операция" требуемыми дополнительными полями, специфичными для проекта "Финансы" (в нашем примере это единственное поле **Сумма**).

```
Extends ОсновыУчета.ОсновныеДокументы.Операция;  
Field Сумма: Число;  
End;
```

Внимание! При использовании атрибута **Extends**, ключевое слово **Document** и название документа должны отсутствовать в описании. Фактически, это означает, что новый класс не создается. Вместо этого в рамках текущего (лицевого) проекта модифицируется класс документов, наследуемый из подпроекта (в нашем случае это класс "Операция" из группы "ОсновныеДокументы" подпроекта "ОсновыУчета").

В результате расширения класса документов те поля, что были определены в базовом классе (в подпроекте) содержат общие данные, разделяемые подпроектом и лицевым проектом. Более того, конкретные экземпляры документов исходного и модифицированного классов - это суть одни и те же объекты, только поля, добавленные в процессе расширения в лицевом проекте, недоступны из подпроекта.

Расширение (через **Extends**) можно использовать только применительно к документам подпроектов из главного (обычно - лицевого) проекта! Расширить документ текущего проекта нельзя.

Если какой-либо проект, в котором объявлено расширение класса документов из подпроекта, в свою очередь используется в качестве подпроекта третьего проекта, то расширенные поля становятся доступными и из этого, третьего, проекта, причем имя класса документов по-прежнему не изменяется - используется имя, данное в подпроекте самого нижнего уровня. Иными словами, все расширенные классы записей подпроектов "видятся" из лицевого проекта в том объеме, который определен подключенными к данному проекту подпроектами. Например, если предположить, что рассмотренный выше подпроект "ОсновыУчета" используется неким проектом "Торговля", то класс Операция будет "виден" из него в урезанном виде - без полей, добавленных в проекте "Финансы".

Расширенную запись можно повторно расширить в проекте еще более высокого уровня.

Следующий важный элемент заголовка документа - строка после знака равно, задающая название основной таблицы и ключевое поле в ней. Предварительно необходимо объяснить, что Студия по умолчанию (если

данный элемент опущен) автоматически генерирует имя основной таблицы (путем транслитерации с русского, если имя документа было на русском), и служебное поле DocID целого типа в качестве ключевого. Однако программист имеет возможность явно указать имя таблицы:

```
Document Операция = OpTable;
```

Здесь приведен заголовок класса документов "Операция", который будет иметь основную таблицу **OpTable**.

Для [внешних документов](#) (то есть документов, подключаемых на чтение из сторонних баз данных) требуется дополнительно указать и ключевое поле, выполняющее роль стандартного поля DocID:

```
Document Операция = OpTable(Key:String);
```

Здесь приведен заголовок класса документов "Операция", который будет подключаться из внешней базы данных, где его основная таблица имеет имя **OpTable** с ключевым полем Key строкового типа.

Важно отметить, что вне зависимости от того, какое имя дано ключевому полю, в [фильтрах](#) (в том числе [запросов](#) и [картотек](#)) оно всегда доступно только под псевдонимом DocID.

Явное описание имени таблицы инициирует внутреннюю процедуру которая называется в терминах Студии привязкой к таблице и, как правило, требуется только в случаях подключения к [внешним базам данным](#).

Наконец, последний еще не описанный модификатор **Abstract** позволяет определять так называемые абстрактные классы документов. Такие классы не могут иметь экземпляров документов и для них не создаются таблицы в базе данных. Основное назначение абстрактных документов - выступать в качестве родительских классов для других классов документов. Иными словами абстрактный класс - это своего рода шаблон класса, содержащий набор базовых свойств.

Abstract Document Операция Title "Основа для всех документов";

При необходимости рассмотренные выше модификаторы иерархичности и абстрактности класса записи могут быть указаны не в начале заголовка, а в конце - в специальной конструкции Options. Конструкция начинается с ключевого слова **Options** (либо **Опции**), после которого в квадратных скобках, через запятую, перечисляются модификаторы:

Ключевое слово	Синонимы
Hierarchical	Иерархический
Abstract	Абстрактный, Абстрактная
RefCount	СчетчикСсылок, ReferencesCounter

Все модификаторы кроме **RefCount** были описаны выше. **RefCount** используется для поддержки контроля целостности ссылок для данного типа записи. Контроль осуществляется путем введения счетчика ссылок на каждую запись, причем счетчик фактически хранится в самой записи в специальном служебном поле, которое добавляется в модель данных при указании данной опции. Включение данной опции существенно ускоряет безопасное удаление записей, но замедляет все остальные операции (добавление, редактирование).

Конструкция **Options** может располагаться в заголовке документа как до, так и после названия (**Title**). Пример:

```
Document Субъект Options [Abstract,Hierarchical];
```

За заголовком документа следуют описания его полей и структур.

При работе с базами данных одной из наиболее часто выполняемых задач является поиск нужных записей по заданному критерию. Задача ускорения поиска записей становится особенно актуальной при работе с большими базами, с которыми оперируют прикладные проекты, разработанные программным средством Студия.

В настоящее время наиболее эффективным методом для решения этой проблемы является метод индексирования. Он заключается в том, что для некоторых особенно часто используемых полей записи создаются, так называемые, индексы, которые позволяют ускорить поиск и фильтрацию записей. Причем, можно создавать индексы, основанные на одном поле или на нескольких полях. Индексирование нескольких полей позволяет различать записи, имеющие одинаковое значение первого поля и т.д.

В программе применяются как служебные индексы, добавляемые автоматически (их нельзя удалять), так и пользовательские индексы, описанные в соответствии с синтаксисом языка Mtl или добавленные пользователем вручную в режиме сессии на странице "[Структура](#)" системного диалога "Просмотр записей".

В описании структуры записи в языке MTL для описания индексов используется синтаксис на русском языке:

```
Индексировано [ИмяИндекса] По <Поля> [Уникальное];
<Поля> = <Поле>[+|-][, <Поля>]
```

или на английском языке:

```
Indexed [Indexname] By <Fields> [Unique];
<Fields> = <Fieldname>[+|-][, <Fields>]
```

Где:

- **ИмяИндекса|Indexname** - необязательное логическое имя индекса;
- **Поля|Fields** - список полей записи, перечисленных через запятую (","), по которым будет строиться индекс;
- **знак "+"** - означает, что индекс будет строиться по возрастанию, причем, знак + можно не указывать;
- **знак "-"** - означает, что индекс будет строиться по убыванию;
- **Уникальное|Unique** - необязательное ключевое слово, указывающее, что индекс будет уникальным по совокупности полей, перечисленных в конструкции **Поля|Fields**.

Внимание. Индекс всегда будет строиться только на те поля, которые явно указаны в описании. Если в индекс требуется добавить служебные поля, то их следует в явном виде перечислить в конструкции **Поля|Fields**.

Пример:

```
Indexed By Сумма; -- индекс по сумме
Indexed Summa By Сумма-, DocID+; -- именованный индекс по сумме и DocID
Indexed By Сумма Unique; -- уникальный индекс по сумме
```

Индексы так же, как и уникальные ключи можно строить как в контексте основной таблицы (если они описаны в записи), так и в контексте структур (если они описаны в структуре **Struct**).

При проектировании информационной базы, как правило, приходится искать компромисс между слишком большим и слишком малым количеством индексов. Хотя индексы позволяют ускорить операции с базой данных, они в то же время требуют дополнительного места (и иногда могут достигать весьма внушительных размеров).

Для задания индексов допускается также старый синтаксис с использованием директивы **Index|Индекс**:

```
Index <ИмяПоля1> [, <ИмяПоля2>...] IndexName ;
Индекс <ИмяПоля1> [, <ИмяПоля2>...] ИмяИндекса;

Здесь:
ИмяПоля1, ИмяПоля2,... - имена полей, по которым строится составной индекс;
ИмяИндекса - логическое имя индекса.
```

Для каждой директивы система строит составной индекс с учетом всех перечисленных в ней полей. В простейшем случае, индекс может создаваться по одному полю:

```
Index Номер;
```

или по нескольким полям, например:

```
Index Номер, Дата;
```


Наряду с [простыми однозначными полями](#), которые могут содержать лишь одно значение указанного типа, в записи часто требуется сохранять несколько однотипных значений, имеющих общий смысл (например, перечень мест работы сотрудника). Для этих целей используются так называемые многозначные поля, к которым относятся *массивы* и *периодические поля*.

Поля-массивы описываются с помощью модификатора **Array**.

```
Field МестоРаботы: String Array Integer;
```

Для каждого массива заводится специальное индексное поле (оно предназначено для индексации значений массива, так как массив может быть упорядочен). Имя индекса можно при желании указать непосредственно после ключевого слова **Array**, если же оно опущено, то по умолчанию берется имя **Index**. Т. е. вышеприведенный пример эквивалентен следующей записи:

```
Field МестоРаботы: String Array Index: Integer;
```

После имени индексного поля (если оно задано) или сразу после слова **Array** (если имя индексного поля опущено) должен быть описан тип индексного поля. В качестве индекса для массива допускается использовать лишь один тип - Integer.

В конце фрагмента описания, относящегося к индексному полю, после знака равенства разрешено в явном виде указывать имя, под которым индексное поле будет храниться в физической записи базы данных.

```
Field МестоРаботы: String Array Порядок: Integer = JOBS;
```

Другой тип многозначного поля - периодическое поле - описывается с помощью ключевого слова **Period**. В отличие от полей-массивов эти поля должны иметь индексы типа Date (чисто теоретически они могли бы быть любого типа, на котором определены отношения больше/меньше: String, Integer, Numeric, Boolean, Date). В остальном периодическое поле аналогично массиву. Чаще всего периодическое поле используется для ведения истории изменения некоторого значения во времени.

Например:

```
Field НалогСПродаж: Numeric Period История: Date;
```

Механизм периодических полей картотеки во многом сходен с [общими переменными](#).

Многозначные поля реализуются на физическом уровне с помощью подчиненных таблиц, в которых хранятся пары индекс-значение.

Общий синтаксис описания поля записи выглядит следующим образом:

```
Field <ИмяПоля>: [Inherited|Унаследованный] <ТипПоля>
  [= [ <ИмяТаблицыОтличнойОтОсновнойТаблицы> . ]<ИмяПоляВФизЗаписи>]
  [Array|Period [ <ИмяИндекса> : ] <ТипИндекса> [= <ИмяПоляВФизЗаписи>]]
  [Title "<Примечание>"];
```

Допускается использование следующих русскоязычных синонимов:

Ключевое слово	Синоним
Field	Поле
Array	Массив
Period	Период
Title	Название

Описание начинается с ключевого слова **Field**, после которого задаются имя поля и его тип. В качестве имени поля может выступать любая строка, отвечающая правилам записи идентификаторов в программе. Ключевые слова **Array|Массив** и **Period|Период** используются для описания, так называемых [многозначных полей](#). Ключевое слово **Title** позволяет указать произвольный комментарий для поля:

```
Field НалогСПродаж: Numeric Period История: Date Title "История ставки налога с продаж";
```

Тип поля - это любой из базовых типов Студии: Integer, Numeric, Logical, String, Date. Разрешается использовать синонимы:

Тип	Синонимы
String	Строка
Integer	Целое, Целый
Numeric	Real, Число
Logical	Boolean, Логическое, Логический
Date	Дата

Для строкового типа разрешено указывать в круглых скобках длину строки, например:

```
String(32);
```

Если длина не задана явно, она устанавливается по умолчанию равной специальному значению, указанному в файле настроек баз данных для конкретного типа СУБД. Как правило, значение по умолчанию установлено равным 250 символам. Рекомендуется явно задавать длину более коротких строк (когда заранее известен максимальный размер), так как это позволяет экономить память.

Компилятор MTL распознает следующие предопределённые константы для задания длины строкового поля:

- **MaxStr | МаксСтр** - максимальный допустимый для используемого сервера размер строкового поля;
- **MaxIndexedStr | МаксИндСтр** - максимальный допустимый для используемого сервера размер индексируемого строкового поля;
- **MaxUniqueStr | МаксУникСтр** - максимальный допустимый для используемого сервера размер уникально индексируемого строкового поля.

Кроме того, возможно указание типов **Text (Текст)**, **Image (Изображение)**, **BinaryObject (ДвоичныйОбъект)** и **OleDocument (OLEДокумент)**, которые, соответственно, декларируют поля для хранения текста (тето-поле), изображения, двоичного объекта (любой файл) и произвольного OLE-документа (документа внешнего приложения, например MS Word). Более того, возможно указание любого типа, производного от класса [Запись](#).

Последний случай весьма важен, так как позволяет организовывать ссылки из одного документа на другой, причем они могут быть разных типов.

```
Field Сотрудник: ФизЛицо;
```

Здесь поле сотрудник имеет тип **ФизЛицо**, то есть является ссылкой на запись класса **ФизЛицо**. В данном случае в том же MTL-файле, где описано поле **Сотрудник**, должен быть также описан и класс документов **ФизЛицо**. Если поле должно ссылаться на запись другой группы или даже другого проекта (подпроекта), то в качестве типа указывается полностью квалифицированное имя, например:

```
Field Сотрудник: Подпроект.ОсновныеДокументы.ФизЛицо;
```

При необходимости перед именем класса записи добавляется модификатор **Inherited (Унаследованный)**, что

означает, что поле может ссылаться не только на запись указанного класса, но и любого класса, производного от указанного. Например, в поле F1

```
Field F1: Унаследованный Накладная;
```

допускается хранить ссылки на записи любых классов, производных от класса Накладная и самого класса Накладная. Следующее описание

```
Field Бумага: Унаследованный Документ;
```

задает поле, в котором можно хранить ссылку на документ любого класса проекта. Исключение составляют некоторые из так называемых [внешних классов документов](#), подключаемых из баз данных других программ. Если во внешнем классе ключевое поле (аналог [служебного поля DocID](#)) не целого типа, то ссылки на такие документы не допускаются. Для внутренних классов записей, полностью поддерживаемых ядром системы, таких ограничений нет.

Хранить в поле записи объект класса **Счет** нельзя. Однако, если это необходимо, то следует описать поле строкового типа и записывать туда идентификатор счета, который легко преобразуется непосредственно в счет с помощью бухгалтерской функции [СчетПоИмени / GetAccountByName](#).

Вот еще несколько описаний простых полей:

```
Field Зарплата: Число;
```

```
Field ФИО: Строка(80);
```

Имя поля (Зарплата, ФИО) является, по сути, логическим именем, под которым данные доступны из прикладного проекта. На физическом уровне, то есть в таблице базы данных, поле, скорее всего, получит другое имя, так как по умолчанию такие имена генерируются системой автоматически (на основе некоторых внутренних правил). Разработчик может в случае необходимости явно указать имя, под которым поле будет храниться в базе данных. Для этого достаточно указать требуемое имя после знака равенства:

```
Field Зарплата: Число = Salary;
```

По умолчанию все поля располагаются в основной таблице, название которой Студия либо также генерирует автоматически, либо устанавливает в соответствии с тем, как оно было определено в [заголовке](#).

Однако, если разработчику необходимо, чтобы поле располагалось в другой таблице базы данных, он может задать имя таблицы явно:

```
Field Зарплата: Число = MOLTBL.Salary;
```

При этом следует иметь в виду, что все таблицы одного документа должны быть расположены в одной физической базе данных.

В состав записи, помимо простых или многозначных полей, могут входить и структуры (подтаблицы). Структура - это поименованный набор логически связанных полей, входящих в состав записи. Синтаксис ее определения во многом схож с описанием поля. Начинается описание структуры с заголовка:

```
Struct <ИмяСтруктуры>  
  [ = <ИмяТаблицыОтличнойОтОсновнойТаблицы> ]  
  [Array|Period [<ИмяИндекса> :] <ТипИндекса> [= <ИмяПоляВФизЗаписи>]]  
  [Title "<Название структуры>"];
```

Заголовок записывается на отдельной строке и включает следующие ключевые слова (записанные на русском или английском языке):

- **Struct|Структура** - ключевое слово, с которого начинается описание структуры. За ним следует *уникальное для данного класса записей имя структуры*;
- **=** - необязательный знак равенства, после которого указывается имя таблицы. Таблица является *основной структурной таблицей*, если имя таблицы опускается при описании полей внутри структуры в привязке поля. Как правило, явное описание имени таблицы требуется только в случаях подключения к [внешним базам данным](#).
- **Array|Массив** или **Period|Период** - ключевые слова, используемые соответственно для описания структуры-массива (подтаблицы) или периодической структуры, значение полей которой зависит от даты. Для периодической структуры после тип данных Дата|Date, а для структуры-массива - Целое|Integer.
- **Title|Название** - необязательное ключевое слово, за которым следует название структуры, записанное в кавычках;

На отдельных строчках после заголовка структуры следует блок, содержащий описание полей, входящих в данную структуру, а завершается описание структуры ключевым словом **end** с точкой с запятой.

Пример описания структуры-массива

```
Record Накладная Title "Товаро-Транспортная накладная";  
  Field Дата:Date;  
  Field Номер:Integer;  
  Field Корресп:ЮрЛицо; -- ссылка на запись типа "ЮрЛицо"  
  Field Примечание:String;  
  
  Struct Позиции Array Integer Title "Табличная часть накладной";  
    Field Index:Integer;  
    Field Товар:Продукция; -- ссылка на карточку товара с названием и пр.  
    Field Цена:Real;  
    Field Количество:Real;  
    Field Сумма:Real;  
  end;  
end;
```

В приведенном примере в записи типа "Накладная" определяется многозначная структура **Позиции**, содержащая перечень товаров (с их атрибутами).

В зависимости от того, является ли структура многозначной или нет, она хранится на физическом уровне либо в подчиненной таблице, либо в основной.

Структура может в свою очередь содержать структуру, причем уровень вложенности структур теоретически не ограничен, однако, следует иметь в виду, что большая вложенность может приводить к замедлению работы программы.

Внутри структуры-массива или периодической структуры может быть описано поле с именем **Index** и с типом, зависящим от вида структуры. Если структура описана как массив, то поле **Index** должно быть типа **Integer**, если структура - периодическая, то **Index** должно быть типа **Date**. Если поле **Index** указано в структуре, то это дает возможность обращаться к нему и получать (только читать) значения индекса для каждого элемента структуры. Если данное поле не указано в структуре, то на физическом уровне соответствующее поле все равно создается, однако, доступ к нему из программы на языке ТБ.Скрипт возможен только с помощью метода [ВзятьПоле](#) базового класса.

MTL-описание структуры разнообразных записей, помимо набора полей, которые перечисляются после ключевого поля **Field|Поле**, может также содержать, так называемые описательные поля. Использование описательных полей позволяет получить развернутое описание записи, например, с целью обеспечения идентификации различных записей.

Внимание! Если описательные поля не назначены и у документа запрашивается описание, то результатом будет значение внешнего ключа, если он задан для этого класса записей, иначе - значение служебного поля [ExtID](#) записи (документа).

Рассматриваемые поля описываются следующим образом:

RecordDescription|ОписаниеЗаписи "Список полей через , или ;"

В соответствии с синтаксисом языка MTL описательные поля указываются после [заголовка документа](#) (см. пример ниже). Их описание начинается с ключевого слова **RecordDescription|ОписаниеЗаписи**, за которым следует список имен полей, разделенных символом ",", или ";", например:

```
RecordDescription Поставщик, Номер, Дата;
```

В списке можно использовать такие поля любого типа, например, Имя, Наименование, Название, Комментарий, Дата и другие, которые содержат отличительные сведения о записи, помогающие ее идентифицировать.

Пример описания

```
record Банк inherited СИС2.Базовая.БазоваяЗапись title "Банковские реквизиты";
  Field Имя      :СИС2.Базовая.ТипИмя      title "Наименование банка";
  Field КСчет    :СИС2.Базовая.ТипКорСтрока title "Корреспондирующий счет";
  Field БИК      :String(9)                title "БИК";
  ....
  RecordDescription Имя;
end;

Document Регион inherited ИерархическийСправочныйДокумент;
  Field Название :String(60);
  RecordDescription Название;
end;
```

Для доступа к значению описательного поля в языке ТБ.Скрипт в классе *Запись* имеется свойство [RecordDescription](#), которое возвращает строковое значение, содержащее значения всех полей по порядку их описания. Причем, если поле не является строковым, то его значение приводится к строковому.

В программе реализованы [системные](#) и [пользовательские](#) ограничения. Пользовательские ограничения накладываются на конкретную запись непосредственно самими пользователями-разработчиками в режиме проектирования. К ним относятся:

- ограничения, описываемые в файлах *.mtl;
- ограничения, задаваемые с помощью программного интерфейса класса [Constraint](#).

При добавлении и изменении пользовательских ограничений осуществляется проверка на синтаксическую корректность устанавливаемого фильтра. В случае обнаружения ошибок в фильтре выдается подробное диагностическое сообщение.

Синтаксис задания ограничений, описываемых на языке MTL

Ограничения, введенные для конкретной записи на языке MTL, действуют для всех пользователей, использующих его, и описываются следующим образом:

Constraint <Условие> on <Событие> Message <Текст сообщения>

Ключевые слова, используемые в синтаксисе:

- **Constraint** - ключевое слово, за которым следует описание условия ограничения <Условие>, составленное по правилам фильтров для запросов. В условии ограничения могут быть использованы макросы. В настоящее время используется один предопределенный макрос
\$CloseDate(<ИмяОбластиУчета>).

Этот макрос возвращает конечную дату [закрытого периода](#) для указанной области учета.

- **on** - ключевое слово, после которого указывается имя события <Событие>, при наступлении которого выполняется ограничение:
 - Post** - при записи документа (записи);
 - Edit** - при редактировании записи, т.е. проверяется версия записи до редактирования;
 - Delete** - при удалении записи;
 - Undelete** - при восстановлении записи.
- **Message** - ключевое слово, за которым записывается сообщение <Текст сообщения>, выдаваемое пользователю при нарушении условия ограничения.

Системные ограничения

Системные ограничения - это те, которые накладываются самой системой, без участия пользователей. Пока, системные ограничения действуют только на сервере данных. Система автоматически добавляет ограничения для каждой записи (документа), по которой построен журнал, но только в том случае если этот журнал относится к [области учета](#), по которой разрешена операция [закрытия периодов](#). Ограничение проверяется при каждой модификации записи и, если запись попадает в закрытый период *хотя бы по одной области учета*, то ее модификация становится невозможной.

Пример

```
Document Накладная inherited Опердок
  Title "Товарно-транспортная накладная";
  Field DocType :Integer Title "Тип документа";

Struct Товары array Integer Title "Товарные позиции";
  Field Услуга      :Logical;
  Field Товар       :Справочники.Товар;
  Field Цена        :Numeric;
  Field Количество  :Numeric;
  Field Сумма       :Numeric;
  Field MYDummy     :String;
end;

Field Назначение :Integer;
Field F1 :Integer;

constraint ЗадайДату
  condition "not Проведен or Дата <> nil"
```

on Post, Edit, Undelete

Message "Должна быть задана дата проведения";

RecordDescription Номер, Дата;
end;

В любом документе в неявном виде всегда присутствует несколько служебных полей, заполняемых самой системой. Часть служебных полей является обязательной: такие поля всегда присутствуют в записях любого класса. Прежде всего, это поле **DocID** целого типа (если иное не указано в явном виде в MTL-описании документа), содержащее уникальный в контексте таблицы номер документа (система гарантирует, что в таблице нет двух документов с одинаковыми значениями **DocID**).

Поле **ExtID** содержит строковое значение - идентификатор документа, уникальный в глобальном масштабе. При создании записи система сама генерирует **ExtID** из имени компьютера, текущего времени и других параметров, в том числе случайных - это гарантирует, что во всех где-либо установленных информационных базах не может встретиться двух документов с одним и тем же значением **ExtID** (если только это не копии одного документа).

Служебные поля типа "дата": **CreateDate**, **UpdateDate** и **ModifyDate**. Поле **CreateDate** содержит время создания конкретной записи. Поля **UpdateDate** и **ModifyDate** хранят время последнего изменения записи: первое из них заполняется при изменении, произведенном пользователем, в то время как второе поле дополнительно фиксирует время изменения записи в случае выполнения служебных операций самой системой, в частности в результате репликации. Иными словами, **ModifyDate** изменяется всегда, а **UpdateDate** - только в результате действий, обусловленных интерактивной работой пользователя. Обычно данные поля хранят равные значения, что означает, что запись в последний раз модифицировалась пользователем, а не системой.

С точки зрения внутреннего устройства Студии поле **ModifyDate** необходимо для того, чтобы бухгалтерское ядро системы постоянно поддерживало расчетные данные в актуальном состоянии, в то время как **UpdateDate** - дает возможность проводить аудит действий с документами.

Поле **ModifyDate** создается всегда, а поля **CreateDate** и **UpdateDate** должны быть явно описаны в MTL-файле, если в проекте требуется информация о датах операций с документами. По умолчанию они не создаются.

Если класс документов был описан с модификатором **Hierarchical**, то добавляются служебные поля **GroupDoc** и **IsGroup**. Поле **IsGroup** целого типа определяет, является данный документ группой или обычным документом. Значение (-1) соответствует группе, а 0 - простому документу. Следует иметь в виду, что поле **IsGroup** записи (таблицы БД) транслируется системой в свойство **IsGroup** объекта класса [Запись](#), создаваемого на основе этой записи БД, причем свойство **IsGroup** имеет логический тип.

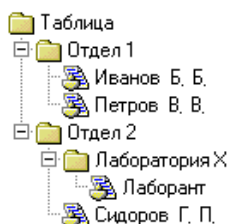
Поле **GroupDoc** того же типа, что и **DocID** (по умолчанию, целого типа), содержит идентификатор документа, определяющего группу, в которую входит текущий документ. Например, пусть описан класс документов "Сотрудник":

```
Hierarchical Document Сотрудник;  
    Field ФИО: Строка;  
End;
```

Если в таблице сотрудников находятся следующие документы:

ФИО	DocID	IsGroup	GroupDoc
Отдел 1 1	TRUE	nil	
Иванов В.В.	12	FALSE	1
Отдел 2 5	TRUE	nil	
Сидоров Г.П.	8	FALSE	5
Петров В.В.	4	FALSE	1
Лаборатория X	32	TRUE	5
Лаборант	10	FALSE	32

то это эквивалентно такой иерархической структуре:



В иерархических классах документов допускается описывать специальное поле **GroupPath** строкового типа. В это поле система автоматически будет записывать полный путь любой записи в иерархии записей. Путь

записывается в виде цепочки значений идентификаторов (**DocID**), указанных через точку. Формальная запись:

```
<IDValue>.[<IDValue>.]
```

то есть преобразованные в строку числа следуют разделенными точками. В конце выражения - всегда точка. Например, путь для записи "Лаборант" из вышеприведенной схемы будет "5.32.10."

Специальное поле **Deleted**, по умолчанию, целого типа хранит признак того, удалена ли текущая запись (здесь следует напомнить, что в системе используется так называемое "мягкое" удаление, когда удаленные записи вначале лишь помечаются особым образом, а их физическое удаление выполняется операцией [сборка мусора](#)). Поле **Deleted** имеет значение ноль, пока запись не удалена, и некоторое положительное значение, если запись удалена. Тип данного поля совпадает с типом поля **DocID**, который можно менять с помощью MTL-описания. Следует иметь в виду, что поле **Deleted** записи (таблицы БД) транслируется системой в свойство **Deleted** объекта класса [Запись](#), создаваемого на основе этой записи БД, причем свойство **Deleted** имеет логический тип.

Дополнительно в MTL-файле есть возможность описать поля с зарезервированными именами:

```
CreateUser :String[(<ДлинаСтроки>)];
```

```
UpdateUser :String[(<ДлинаСтроки>)];
```

причем длину данных полей указывать хоть и не обязательно, но настоятельно рекомендуется. Связано это с тем, что имя пользователя, как правило, гораздо короче, чем длина строкового поля, принимаемая по умолчанию (то есть если размер поля не указан явно). Таким образом, указание некоторого приемлемого значения (например, в диапазоне от 30 до 60) символов позволяет избежать излишней траты памяти.

Если одно или оба данных полей описано, то они автоматически заполняются и доступны только на чтение. В первое поле система заносит имя пользователя, создавшего документ, а во второе - имя пользователя, который последним редактировал документ. По данным полям можно устанавливать фильтр и выполнять все остальные операции (кроме записи), разрешенные для остальных полей. Следует внимательно относиться к необходимости использования этих полей и их размеру, так как с их введением возможно существенное увеличение размера таблицы.

Описанные пользователем поля записей почти всегда транслируются системой в одноименные свойства объектов соответствующего класса документов, производных от класса **Запись**. Исключение составляют случаи, когда имя поля совпадает с названием свойства класса **Запись**. Тогда свойство класса перекрывает поле записи, и чтобы получить к нему доступ необходимо использовать метод **GetField** класса [Запись](#). Служебные поля также, как правило, транслируются в свойства класса **Запись**, однако в некоторых случаях (например, как в рассмотренных выше полях **IsGroup** и **Deleted**) тип свойств может отличаться от типа полей.

Часто имеет место ситуация, когда значения некоторых полей документа (или сочетания значений нескольких полей) в силу своей специфики должны быть уникальными для таблицы. Достигается это с помощью механизма уникальных ключей.

Уникальный ключ - это непустое множество полей (то есть как минимум одно поле или более), значения которых в указанном сочетании не могут совпадать у двух или более записей одной картотеки (таблицы).

Так, служебное поле **DocID**, которое не требует явного описания (хотя и может быть описано принудительно), является уникальным ключом. Если разработчику необходимо обеспечить уникальность каких-либо других полей, он может воспользоваться директивой **Unique (Уникальные или Уникальное)**:

```
Unique <ИмяПоля> [ , <ИмяПоля>...];  
Уникальное <ИмяПоля> [ , <ИмяПоля>...];
```

Все, перечисленные после ключевого слова **Unique** поля образуют уникальный ключ, то есть система не даст пользователю сохранить два документа с одинаковыми значениями в данных полях. Вот пример для одного поля:

```
Unique НомерСтраховогоПолиса;
```

А теперь пример составного ключа:

```
Unique Контрагент, ИсходящийНомерПисьма;
```

В последнем примере задан уникальный ключ из двух полей: **Контрагент** и **ИсходящийНомерПисьма**. Тогда при вводе информации в картотеку будет проверяться, чтобы никакие две записи не имели одновременно одинаковых названий контрагента и номеров входящей корреспонденции от данного контрагента.

Уникальными могут быть только скалярные (немногозначные) поля. Нельзя указывать в одной директиве **Unique** одно и то же поле более одного раза.

Интересно, что уникальности полей можно "требовать" как на уровне документа, так и на уровне структур **Struct** (имеются в виду структуры-массивы, описанные с модификатором **Array** или **Period**). В последнем случае необходимо написать директиву **Unique** внутри блока **Struct** и указать в ней поля из числа входящих в данную структуру. Если поле (или сочетание полей) структуры объявлено уникальным, то система запрещает вводить в структуру элементы с совпадающими значениями ключей. Таким образом можно обеспечить уникальность строк в подтаблице документа.

Поля, указанные в директиве **Unique**, должны быть обязательно описаны внутри текущего блока (блока описания документа или блока описания структуры **Struct**). Недопускается в одной и той же директиве **Unique** комбинировать поля из документа и его структур.

Класс записи может иметь несколько уникальных ключей.

MTL-описание начинается с директивы **Group (Группа)** со следующим синтаксисом:

```
Group "<Название-комментарий>" ;  
Группа "<Название-комментарий>" ;
```

После ключевого слова **Group (Группа)** следует заключенное в кавычки название (комментарий), и завершает строку точка с запятой. Следует иметь в виду, что внутри проекта группа идентифицируется по имени MTL-файла, а приведенное в директиве **Group** название используется лишь для уточнения назначения группы (выступает в роли пояснения для разработчиков).

В конце MTL-файла должен находиться парный для директивы **Group** оператор **End**. То есть все описания документов, имеющиеся в MTL-файле, должны быть заключены в блок **Group ... End**.

Один MTL-файл содержит описание только одной группы документов.

После заголовка группы могут следовать необязательные описания нестандартных пользовательских типов, применяющихся в данном MTL-файле. Синтаксис описания типа следующий:

```
Тип <ИдентификаторНовогоТипа> = <СтарыйТип>;  
Тип <ИдентификаторНовогоТипа> = <СтарыйТип>;
```

В качестве идентификатора нового типа может использоваться любая строка, отвечающая правилам записи идентификаторов.

Справа от знака "равно" указывается старый стиль - это может быть любой стандартный тип данных или определенный выше пользовательский тип. Например:

```
Тип Money = Numeric;  
Тип КороткаяСтрока = String(60);  
Тип ФИО = КороткаяСтрока;
```

После определения нового типа его можно использовать в последующих MTL-описаниях. Как правило, введение новых типов используется для получения более краткой и понятной записи. Проиллюстрируем использование предопределенных типов на примере описания полей:

```
Field Salary: Money;  
Field Сотрудник: ФИО;
```

Внимание! Определения нестандартных типов действуют только внутри одного файла MTL, в котором они описаны.

В MTL файле описывается структура документов, которыми оперирует прикладной проект Студии. Физически информация хранится в реляционной базе данных. Процесс преобразования логической структуры документа в реляционные отношения называется *нормализацией*, а обратный процесс, соответственно, *денормализацией*. Управляет процессами нормализации/денормализации сервер данных (сервер приложений). Для документов, которые создаются и модифицируются Студией правила нормализации известны серверу и не требуют уточнений. При подключении внешних баз данных (управляемых другими программами) правила денормализации реляционных таблиц требуется указать явно. Для решения этой задачи и предлагается расширенная спецификация языка MTL.

Основное отличие расширенной спецификации от относительного простого синтаксиса MTL, рассмотренного выше и используемого при проектировании "родных" баз данных Студии, заключается в необходимости *в явном виде прописывать отношения между таблицами физической базы данных*.

Внешней может быть описана только группа документов целиком. Напомним, что в одном MTL-файле всегда описывается только одна группа документов, поэтому все входящие в нее классы документов либо внешние, либо внутренние (собственные). Внешняя группа имеет заголовок вида:

```
External Group "<Название-комментарий>";
```

По сравнению с обычным MTL-описанием здесь добавилось лишь одно ключевое слово - модификатор **External (Внешняя)**.

Часть синтаксических конструкций, определяющих отношения между входящими в группу таблицами физической базы данных уже описывалась в параграфах посвященных [описанию документов](#).

Так, в заголовке документа, в описаниях полей и структур программист может указать (после знака равенства) имя таблицы. Эта привязка к таблице выполняется для собственных таблиц Студии автоматически, но необходима при интеграции с внешними источниками данных (например, базами данных для ПО сторонних производителей).

Кроме того, в описании документа может явно указываться подчиненность таблиц. Действительно, если накладная (см. пример в разделе [Описание документов](#)), имеющая основную часть (таблицу) с реквизитами и подтаблицу с перечнем товаров, хранится во внешней базе данных, то определить взаимосвязь составных частей и обработать такой документ как единое целое невозможно без дополнительных регуляций.

Синтаксис описания внешних подчиненных таблиц следующий:

```
Table <ИмяТаблицыПодчиненной> Joined <ИмяПоляПодчиненнойТаблицы>
[ = [ИмяТаблицыОсновной.]ИмяПоляИзОснТаблицы];
```

Допускается использовать русские синонимы:

Ключевое слово	Синоним
Table	Таблица
Joined	Связана

Таким образом, внешняя таблица описывается с помощью ключевого слова **Table**, после которого указывается имя таблицы, ключевое слово **Joined** и пара полей в основной и подчиненной таблицах, обеспечивающая их взаимосвязь. Имя основной таблицы может быть опущено - в этом случае оно берется из заголовка документа.

Пример:

```
Document Накладная=Invoice(DocId :Integer)
  Title "Товаро-Транспортная накладная";
  Table Common Joined DocId=Invoice.DocId;
  Table InvTab Joined DocId=Invoice.DocId;

  Field Дата           :Date      = Date;
  Field Номер          :Integer    = Number;
  Field Корресп        :ЮрЛицо    = Corresp;
  Field Примечание     :String     = Descr;

  Struct Инфо = Common Title "Общая информация о документе";
    Field ДатаСозд      :Date      = CreateDate;
    Field ДатаПров      :Date      = TransDate;
  end;

  Struct Позиции = InvTab array Index : Integer = Order
    Title "Табличная часть накладной";
    Field Товар         :Продукция = Prod;
```

```

Field Цена           :Real      = Price;
Field Количество     :Real      = Quant;
Field Сумма          :Real      = Sum;
end;
end;

```

Система допускает описание внешнего документа с подтаблицами, даже если в подтаблице отсутствует целочисленное инкрементное поле, например:

```

Record Накладные = "Orders"("OrderNo" : Numeric);
  Table "Items" Joined "OrderNo" = "Orders"."OrderNo";
  ...
  Struct Позиции = "items" period ItemNo : Real = "ItemNo";
  ...
end;
end ;

```

Как видно из этого примера, тип индексного поля в периодической структуре – вещественное число.

Изменение MTL-описания модели данных работающего проекта приводит к необходимости преобразовать имеющиеся данные к новой модели. Поскольку база данных может содержать очень большие объемы информации, провести такое изменение "на лету" не представляется возможным. Реорганизация неизбежно влечет за собой корректировку внутреннего формата хранения данных и, как следствие, необходимость обработки каждой записи, входящей в изменяемую картотеку.

Процесс *реорганизации* базы данных, т.е. применения сделанных в MTL-файле изменений к реальным данным, производится при первой попытке запустить проект (при подключении к информационной базе) после его модификации. Система сама определяет, что изменился один или несколько MTL-файлов и предлагает провести реорганизацию, которая выполняется под управлением мастера. Как правило, пользователю достаточно на всех шагах процесса соглашаться с предложенными по умолчанию параметрами. Время, затрачиваемое на реорганизацию, напрямую зависит от объема данных и мощности компьютера.

Замечание. Требуется, чтобы в момент проведения реорганизации все картотеки и бланки, их использующие, были закрыты. Особенно важно следить за выполнением этого условия при работе с сетевой версией проекта.

Проведение реорганизации - процесс очень ответственный, поскольку в процессе изменения структуры картотек "лишняя" информация физически удаляется. Поэтому необходимо тщательно проверить правильность новой модели данных и убедиться в необходимости реорганизации.

Один MTL-файл описывает логическую группу взаимосвязанных документов. Количество MTL-файлов в проекте не регламентировано и определяется разработчиком, исходя из требований самого проекта.

MTL-описание должно быть заключено в один групповой блок, который начинается с ключевого слова **Group|Группа** и заканчивается ключевым словом **End|Конец** с точкой с запятой. [Заголовок группы](#) описывается в соответствии с требуемым синтаксисом языка.

Внутри группы следуют [описания нестандартных типов](#) и собственно [описания документов \(записей\)](#), включающие название документа, перечень входящих в него полей (название, тип, размер) и некоторые другие сведения.

Следует иметь в виду, что часть описываемых документов может быть связана с внешними документами, расположенными в сторонних базах данных. Как правило, такая ситуация возникает при интеграции прикладного проекта Студии с программами других производителей, установленными у заказчика. Поскольку внешние базы данных функционируют по правилам внешних программ по отношению к Студии, такие базы доступны только на чтение, и описание связанных с ними документов требует введения в MTL-описание дополнительных параметров, которые обычно опускаются для собственных баз данных Студии. В связи с этим мы вначале рассмотрим упрощенный синтаксис MTL-описаний, достаточный для создания полнофункционального, автономного проекта Студии, а затем расширим его необязательными атрибутами, управляющими специфическими функциями MTL.

MTL-файлы компилируются, так же как файлы с текстами программ на языке ТБ.Скрипт, хотя, конечно, компилятор ТБ.Скрипт и компилятор MTL - две разных подсистемы Студии. Это станет очевидным после изучения языка MTL-описания.

На стадии разработки программист оперирует текстовым MTL-файлом, который после компиляции преобразуется в двоичное представление и предопределяет структуру данных на уровне информационных баз и, в конечном счете, на уровне картотек.

Важно отметить, что каждое изменение MTL-файла влечет за собой последующий запуск процедуры реструктуризации базы данных на стадии исполнения проекта. Действительно, MTL-описание напрямую связано с физической структурой данных в базе - управляет ею. Следовательно, при редактировании MTL-файлов для уже существующих проектов необходимо быть особенно осторожным, чтобы не потерять данные.

Как и во всех текстах Студии, при описании модели данных можно использовать [комментарии](#). Комментарий служит для пояснения текста MTL и начинается с двух минусов. Весь фрагмент строки, начиная от двух минусов и до ее конца, не участвует в компиляции.

По аналогии с текстами ТБ.Скрипт, MTL-описание допускает использование так называемых [директив условной компиляции](#), однако, с некоторыми ограничениями. В директивах условных компиляции в mtl-файлах нельзя использовать выражения, а только простые идентификаторы ключей компиляции и константы. Например, для определения ключа компиляции используется директива [#define](#) со следующим синтаксисом:

```
#Определить <Имя> = (<ИзвестноеИмя>|<Константа>) ;  
где Имя - идентификатор определяемого ключа компиляции;  
ИзвестноеИмя - идентификатор определенного ранее ключа;  
Константа - конкретное значение.
```

Затем ключи компиляции могут проверяться с помощью директивы [#if](#):

```
#если [не] (<ИзвестноеИмя>|<Константа>) тогда
```

- в случаях, когда значение ключа или константы имеет логический тип;

```
#если (<ИзвестноеИмя>|<Константа>) ( = | <> ) (<ИзвестноеИмя>|<Константа>) тогда
```

- в случаях, когда используются значения любых базовых типов (то есть разрешено только сравнение на равенство и неравенство).

При создании информационной базы все классы документов, входящие в состав проекта, распределяются по физическим базам данных. Разные классы документов могут быть либо разнесены по разным БД, либо объединены в одну. При этом если документ имеет сложную структуру с многозначными полями, он будет содержать больше одной физической таблицы, и все его таблицы должны быть отнесенными в одну БД. Это важно иметь в виду, если используются специальные возможности языка MTL, позволяющие явным образом задавать имена физических таблиц для классов документов.