

Реализация хранимых процедур и функций

Содержание:

[Урок 1: Реализация хранимых процедур](#)

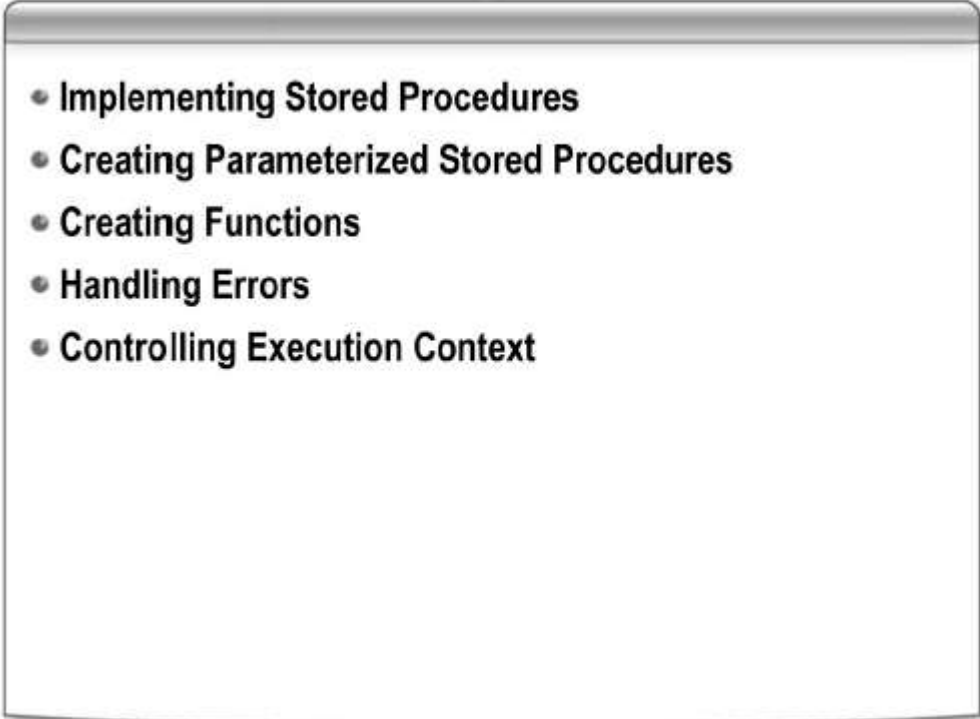
[Урок 2: Создание параметризованных хранимых процедур](#)

[Урок 3: Создание функций](#)

[Урок 4: Обработка ошибок](#)

[Урок 5: Управление контекстом выполнения](#)

Реализация хранимых процедур и функций

- 
- Implementing Stored Procedures
 - Creating Parameterized Stored Procedures
 - Creating Functions
 - Handling Errors
 - Controlling Execution Context

Цели

После завершения этой темы, студенты смогут:

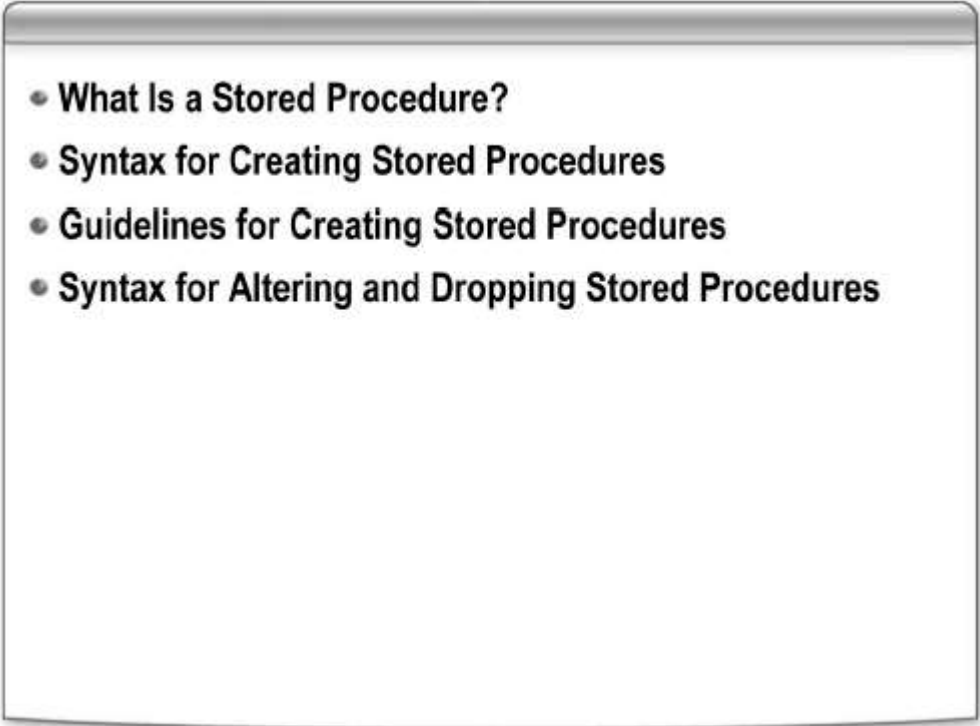
- Реализовать хранимые процедуры.
- Создать параметризованные хранимые процедуры.
- Реализовать функции.
- Обработать ошибки.
- Управлять контекстом выполнения.

Введение

От Вас могут ожидать, как от разработчика базы данных, что Вы будете проектировать и реализовывать логику в базе данных, чтобы воплотить бизнес-правила или согласованность данных с использованием хранимых процедур или пользовательских функций. Или Вы могли бы быть ответственным за изменение и поддержание существующих модулей, написанных другими разработчиками.

Из материалов этой темы Вы узнаете, как создать хранимые процедуры и определяемые пользователем функции. Вы также узнаете, как осуществить структурированную обработку ошибок и управлять контекстом выполнения.

Урок 1: Реализация хранимых процедур

- 
- **What Is a Stored Procedure?**
 - **Syntax for Creating Stored Procedures**
 - **Guidelines for Creating Stored Procedures**
 - **Syntax for Altering and Dropping Stored Procedures**

Цели урока

После завершения этого урока, студенты смогут:

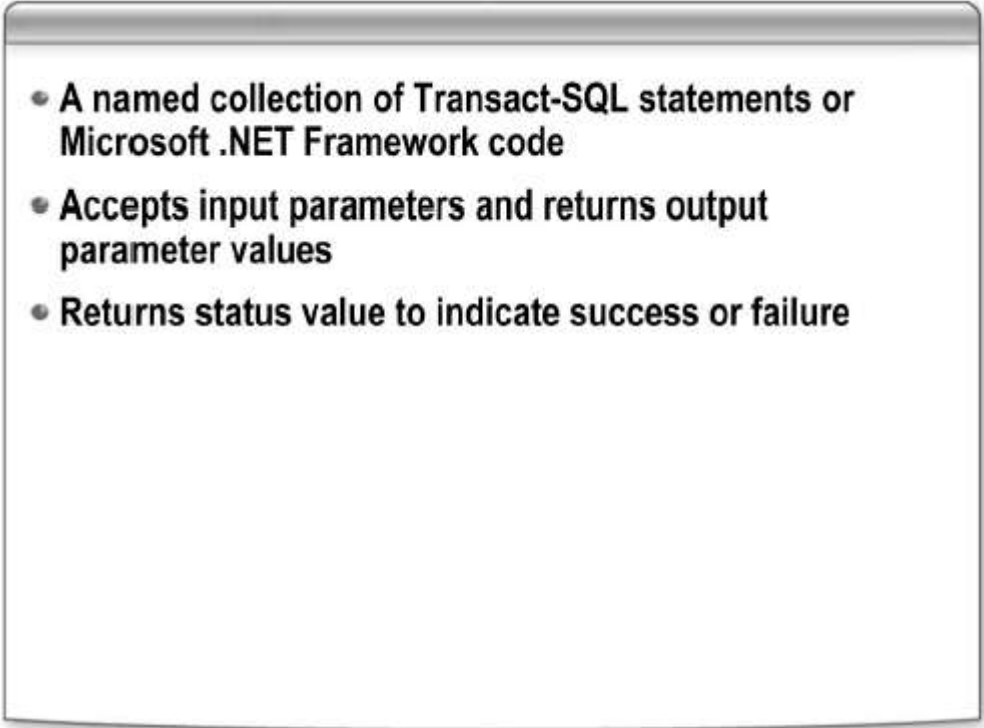
- Определять хранимые процедуры, и описывать цели использования хранимых процедур.
- Описывать синтаксис создания хранимых процедур.
- Описывать лучшие практические приемы создания хранимых процедур.
- Описывать синтаксис изменения и удаления хранимой процедуры.

Введение

Хранимая процедура – это именованная группа операторов Transact-SQL, откомпилированных в единый план выполнения. Хранимые процедуры могут помочь вам в достижении согласованности реализации логики через приложения.

Этот урок описывает хранимые процедуры и какие преимущества они могут предоставить для Ваших приложений базы данных. Вы также узнаете, как изменить или удалить существующие хранимые процедуры.

Что такое хранимая процедура?

- 
- **A named collection of Transact-SQL statements or Microsoft .NET Framework code**
 - **Accepts input parameters and returns output parameter values**
 - **Returns status value to indicate success or failure**

Что такое хранимая процедура?

Хранимая процедура - именованная коллекция операторов Transact-SQL, которая хранится на сервере непосредственно в базе данных. Хранимые процедуры – это метод инкапсулирования повторяющихся задач; они поддерживают пользовательские переменные, условные операторы и другие возможности программирования.

Хранимые процедуры в Microsoft® SQL Server™ подобны процедурам в других языках программирования, в которых они могут:

- Содержать операторы, которые выполняют операции в базе данных, включая способность вызывать другие хранимые процедуры.
- Принимать входные параметры.
- Возвращать значение состояния вызова хранимой процедуры или пакета: успешное или неудачное.
- Возвращать множество значений хранимой процедуры или приложения-клиента в форме выходных параметров.

Преимущества хранимой процедуры

Хранимые процедуры предлагают многочисленные преимущества перед выполнением запросов Transact-SQL. Они могут:

Инкапсулировать прикладные функциональные возможности и создавать прикладную логику многократного использования. Бизнес-правила или политики, формируемые в хранимых процедурах, могут изменяться в одном месте. Все клиенты могут использовать одни и те же хранимые процедуры, чтобы гарантировать согласованный доступ к данным и модификацию.

Оградить пользователей от деталей организации таблиц в базе данных. Если ряд хранимых процедур поддерживают все бизнес-функции, которые пользователи должны выполнять, то пользователи никогда не должны обращаться непосредственно к таблицам.

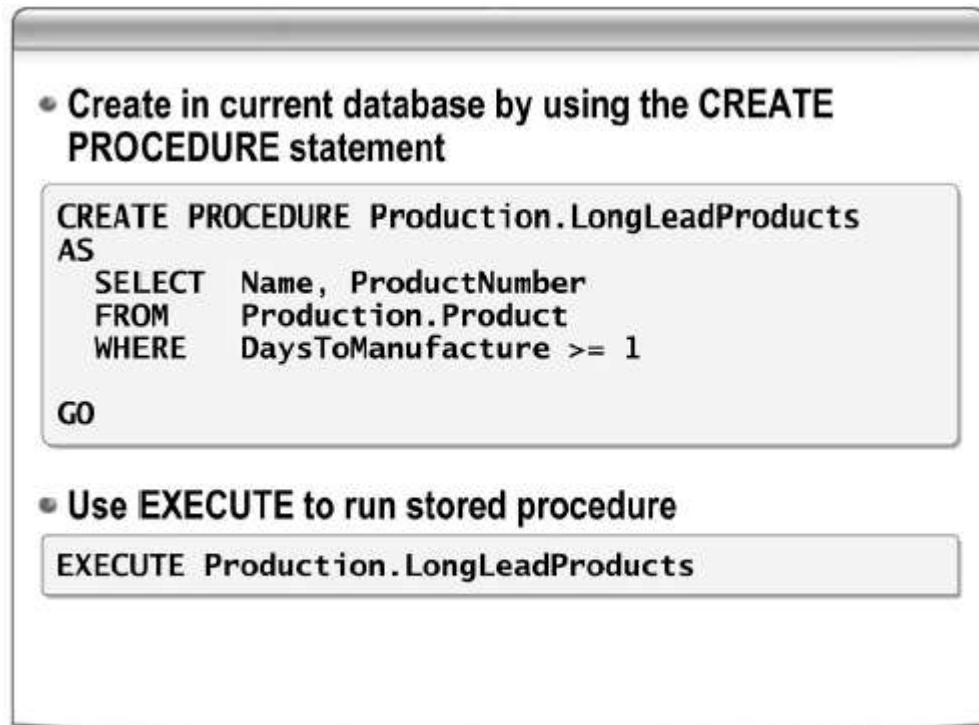
Обеспечить механизмы безопасности. Пользователям можно предоставить разрешение выполнить хранимую процедуру, даже если у них нет разрешения обратиться к таблицам или представлениям, на которые ссылается хранимая процедура.

Улучшить производительность. Хранимые процедуры осуществляют много задач как серия операторов Transact-SQL. К результатам первых операторов Transact-SQL может быть применена условная логика, чтобы определить, какие последующие операторы Transact-SQL должны выполняться. Все эти операторы Transact-SQL и условная логика становятся частью единственного плана выполнения на сервере.

Уменьшить сетевой трафик. Вместо того, чтобы посылать сотни операторов Transact-SQL по сети, пользователи могут выполнить сложную операцию, посылая единственный оператор, который сокращает количество запросов, проходящих между клиентом и сервером.

Уменьшить уязвимость от атак SQL. Использование явно определенных параметров в SQL коде устраняет возможность того, что хакер мог представить внедренные операторы SQL в значениях параметра.

Синтаксис создания хранимых процедур



Введение

Вы создаете хранимые процедуры при использовании оператора **CREATE PROCEDURE**. Хранимые процедуры могут быть созданы только в текущей базе данных – за исключением временной хранимой процедуры, которая всегда создается в базе данных **tempdb**. Создание хранимой процедуры подобно созданию представления. Сначала напишите и проверьте операторы Transact-SQL, которые Вы хотите включить в хранимую процедуру. Тогда, если Вы получаете ожидаемые результаты, создайте хранимую процедуру.

Частичный синтаксис для создания хранимой процедуры

Оператор **CREATE PROCEDURE** содержит много возможных опций, как показано в следующем частичном синтаксисе.

CREATE { **PROC** | **PROCEDURE** } [schema_name.] procedure_name

[{ @parameter [type_schema_name.] data_type }

[**VARYING**] [= default] [[**OUT** [**PUT**]]

[,...n]

[**WITH** <procedure_option> [,...n]

```
AS sql_statement [;][ ...n ]
```

```
<procedure_option> ::=
```

```
    [ ENCRYPTION ]
```

```
    [ RECOMPILE ]
```

```
    [ EXECUTE_AS_Clause ]
```

Дополнительная информация. Для получения дополнительной информации о синтаксисе CREATE PROCEDURE, см. "CREATE PROCEDURE (Transact-SQL)" в SQL Server Books Online.

Пример создания простой хранимой процедуры

Следующий пример показывает, как Вы можете создать простую хранимую процедуру, возвращающую набор строк всех продуктов, производство которых занимает больше чем один день.

```
CREATE PROC Production.LongLeadProducts
```

```
AS
```

```
    SELECT Name, ProductNumber
```

```
    FROM Production.Product
```

```
    WHERE DaysToManufacture >= 1
```

```
GO
```

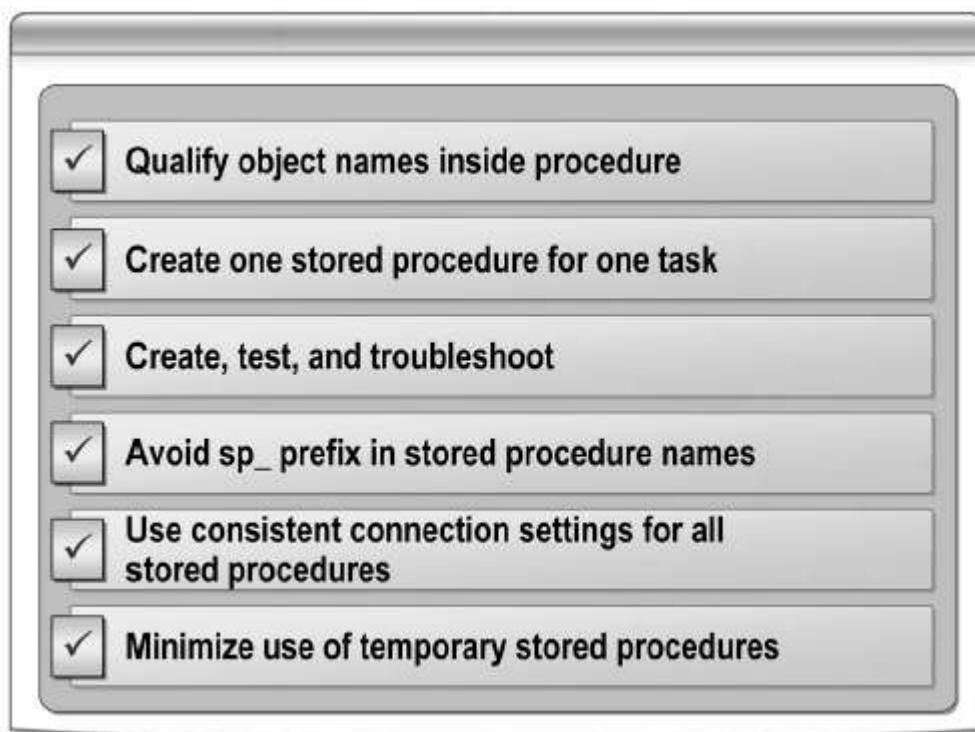
Предыдущий пример создает процедуру по имени **LongLeadProducts** в схеме **Production**. Команда GO включена, чтобы подчеркнуть тот факт, что операторы CREATE PROCEDURE должны быть объявлены в пределах одного пакета.

Пример вызова хранимой процедуры

Следующий пример показывает, как вызвать хранимую процедуру **LongLeadProducts**.

```
EXEC Production.LongLeadProducts
```

Рекомендации для создания хранимых процедур



Рекомендации по хранимым процедурам

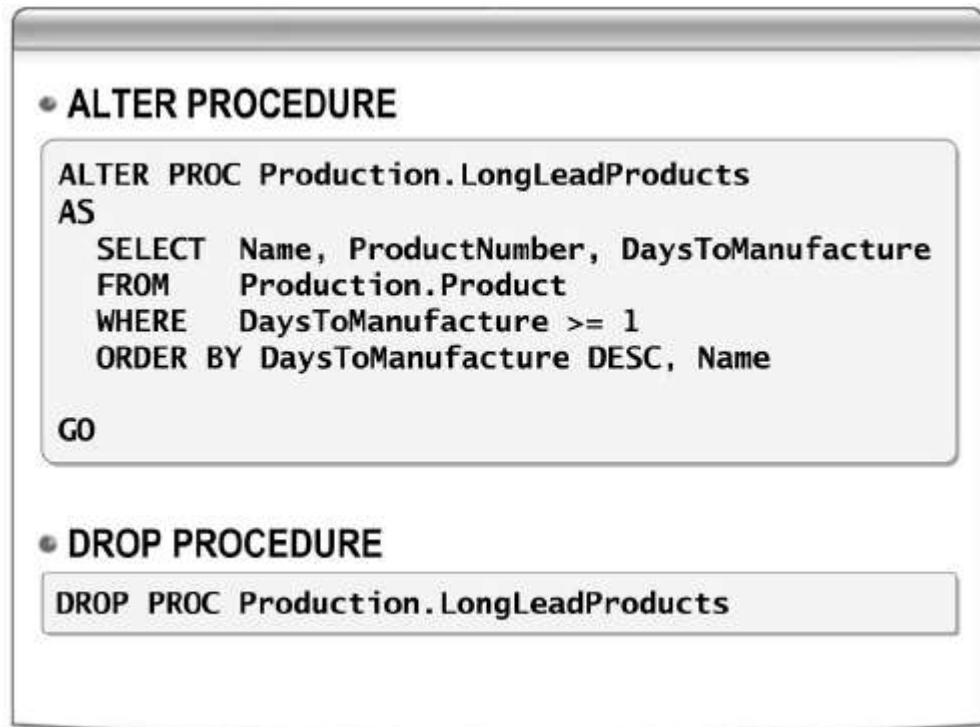
Следуйте следующим рекомендациям, когда Вы создаете хранимые процедуры:

- Квалифицируйте имена объектов в хранимой процедуре с соответствующим названием схемы. Это гарантирует то, что таблицы, представления, или другие объекты из различных схем будут доступны в данной хранимой процедуре. Если имя объекта не квалифицировано, то будет принята по умолчанию схема хранимой процедуры.
- Проектируйте каждую хранимую процедуру для выполнения одной задачи.
- Создавайте, тестируйте, и отлаживайте Вашу хранимую процедуру на сервере, а затем проверьте ее на клиенте.
- Избегайте использовать префикс **sp_**, при именовании локальных хранимых процедур для различения их от системных хранимых процедур. Другая причина избегать префикса **sp_** для хранимых процедур в своей базе данных - избежать ненужных поисков в базе данных **master**. Когда вызывается хранимая процедура с именем, начинающимся с **sp_**, то SQL Server ищет в базе данных **master** прежде, чем будет искать в исходной базе данных.
- Использование те же самые параметры настройки подключения для всех хранимых процедур. SQL Server сохраняет параметры настройки и SET QUOTED_IDENTIFIER и

опции SET ANSI_NULLS когда хранимая процедура создается или изменяется. Эти оригинальные параметры настройки используются, когда выполняется хранимая процедура. Поэтому, любые параметры настройки клиентского сеанса для этих опций SET игнорируются во время выполнения хранимой процедуры.

■ Минимизируйте использование временных хранимых процедур, чтобы избежать заполнения системных таблиц в **tempdb**, эта ситуация может неблагоприятно отразиться на производительности.

Синтаксис для изменения и удаления хранимых процедур



Изменение хранимой процедуры

Хранимые процедуры часто изменяются в ответ на запросы от пользователей или на изменения определений основных таблиц. Чтобы изменить существующую хранимую процедуру и сохранить разрешения, используйте оператор **ALTER PROCEDURE**. SQL Server заменяет предыдущее определение хранимой процедуры при использовании **ALTER PROCEDURE**.

Учитывайте следующие факты, когда Вы используете оператор **ALTER PROCEDURE**:

- Если Вы хотите изменить хранимую процедуру, которая была создана при использовании опции **WITH ENCRYPTION**, Вы должны включить эту опцию в оператор **ALTER PROCEDURE**, чтобы сохранить те функциональные возможности, которые обеспечивает эта опция.
- **ALTER PROCEDURE** изменяет только одну процедуру. Если Ваша процедура вызывает другие хранимые процедуры, то вложенные хранимые процедуры не изменяются.

Пример изменения хранимой процедуры

Следующий пример изменяет хранимую процедуру **LongLeadProducts**, чтобы выбрать дополнительный столбец и сортировать результирующий набор при использовании выражения ORDER BY.

```
ALTER PROC Production.LongLeadProducts
AS
    SELECT Name, ProductNumber, DaysToManufacture
    FROM Production.Product
    WHERE DaysToManufacture >= 1
    ORDER BY DaysToManufacture DESC, Name
GO
```

Удаление хранимой процедуры

Используйте оператор DROP PROCEDURE, чтобы удалить определяемые пользователем хранимые процедуры из текущей базы данных.

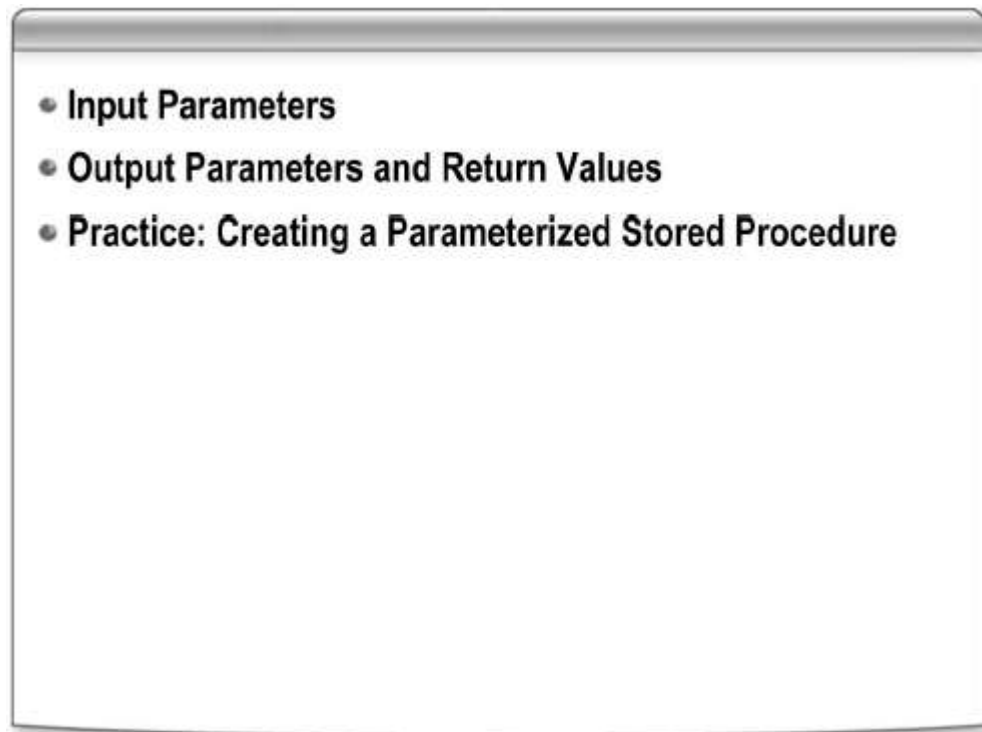
Прежде, чем Вы удалите хранимую процедуру, выполните хранимую процедуру **sp_depends** для определения объектов, которые зависят от этой хранимой процедуры, как показано в следующем примере.

```
EXEC sp_depends @objname = N'Production. LongLeadProducts'
```

Следующий пример удаляет хранимую процедуру **LongLeadProducts**.

```
DROP PROC Production.LongLeadProducts
```

Урок 2: Создание параметризованных хранимых процедур



Цели урока

После завершения этого урока, студенты смогут:

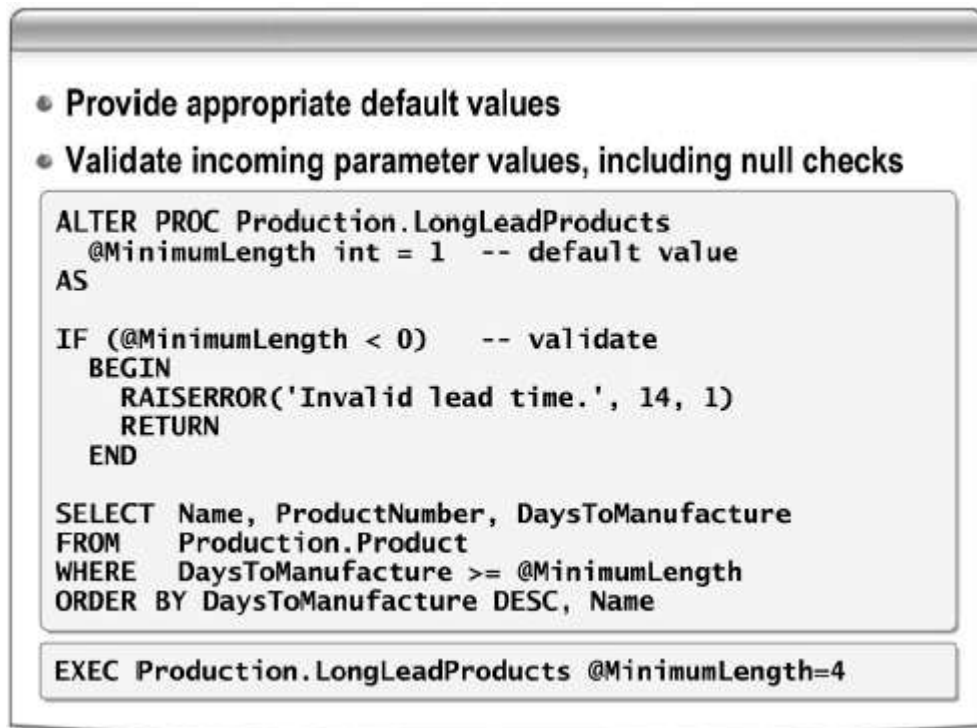
- Описать синтаксис использования входные параметров.
- Описать синтаксис использования выходные параметры и возвращаемых значений.

Введение

Хранимые процедуры более гибки, когда Вы включаете параметры как часть определения процедуры, потому что Вы можете создать более универсальную прикладную логику.

Из этого урока Вы узнаете, как включать входные и выходные параметры и как использовать возвращаемые значения в хранимых процедурах.

Входные параметры



Введение

Хранимая процедура взаимодействует с программой, которая вызывает процедуру, через список до 2100 параметров. Входные параметры позволяют информации быть переданной в хранимую процедуру; тогда эти значения могут использоваться как локальные переменные в процедуре.

Рекомендации по использованию входных параметров

Чтобы определить хранимую процедуру, которая принимает входные параметры, Вы объявляете один или более переменных как параметры в операторе CREATE PROCEDURE.

Рассмотрите следующие рекомендации по использованию входных параметров:

- Обеспечьте значениями по умолчанию соответствующие параметры. Если значение по умолчанию определено, пользователь может выполнить хранимую процедуру, не определяя значение для этого параметра.
- Проверьте правильность всех входящих значений параметра в начале хранимой процедуры для раннего отлавливания отсутствующих и недопустимых значений. Можно включать проверку параметров на null.

Пример использования входных параметров

Следующий пример добавляет параметр **@MinimumLength** для хранимой процедуры **LongLeadProducts**. Это позволяет выражению WHERE быть более гибким, чем ранее показанный, позволяя вызывающему приложению определить, какое время выполнения заказа считается допустимым.

```
ALTER PROC Production.LongLeadProducts
    @MinimumLength int = 1      -- default value
AS

IF (@MinimumLength < 0)      -- validate
    BEGIN
        RAISERROR('Invalid lead time.', 14, 1)
        RETURN
    END

SELECT Name, ProductNumber, DaysToManufacture
FROM      Production.Product
WHERE      DaysToManufacture >= @MinimumLength
ORDER BY DaysToManufacture DESC, Name
```

Хранимая процедура определяет заданное по умолчанию значение **1** параметра так, чтобы вызывающее приложение могло выполнить процедуру, не определяя параметр. Если значение передается для **@MinimumLength**, то оно проверяется для гарантии того, что значение является допустимым для оператора SELECT. Если значение - меньше чем нуль, то возбуждается ошибка, и хранимая процедура немедленно завершается без выполнения оператора SELECT.

Вызов параметризованных хранимых процедур

Вы можете установить значение параметра, передавая значение хранимой процедуре через имя параметра или позицию. Вы не должны смешивать различные форматы, когда Вы указываете значения параметров.

Определяя параметр в операторе EXECUTE в формате *@parameter = значение*, называется *передача по имени параметра*. Когда Вы передаете значения по имени параметра, то они могут быть определены в любом порядке, и можно опустить параметры, которые позволяют null значения или те, у которых есть значение по умолчанию.

Следующий пример вызывает хранимую процедуру **LongLeadProducts** и определяет название параметра.

```
EXEC Production.LongLeadProducts @MinimumLength=4
```

Передача только значений (без имен параметров, для которых они передавались), называется *передача значений по позиции*. Когда Вы определяете только значения параметров, то они должны быть перечислены в порядке, в котором они определены в операторе CREATE PROCEDURE. Когда Вы передаете значения по позиции, Вы можете опустить параметры, где существуют значения по умолчанию, но Вы не можете прервать последовательность. Например, если у хранимой процедуры есть пять параметров, Вы можете опустить четвертый и пятый параметры, но Вы не можете опустить четвертый параметр и определить пятый.

Следующий пример вызывает хранимую процедуру **LongLeadProducts** и определяет параметр по позиции.

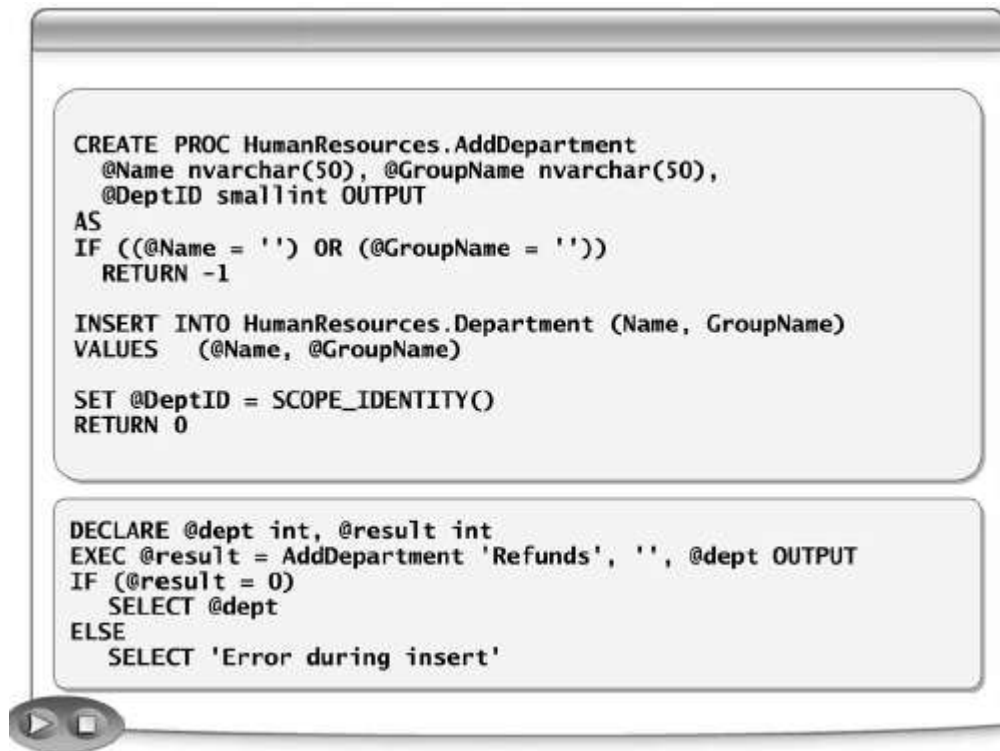
```
EXEC Production.LongLeadProducts 4
```

Использование значения по умолчанию параметра

Если в хранимой процедуре для параметра определено значение по умолчанию, то оно используется, когда:

- Не определено никакое значение для параметра, когда хранимая процедура выполняется.
- Определено ключевое слово DEFAULT как значение параметра.

Выходные параметры и возвращаемые значения



Введение

Хранимые Процедуры могут вернуть информацию вызывающей хранимой процедуре или клиенту с помощью выходных параметров и возвращаемого значения.

Характеристики выходных параметров

Выходные параметры позволяют любые изменения параметра, которые сохраняются в результате выполнения хранимой процедуры, и они будут сохранены даже после того, как хранимая процедура завершит выполнение. Чтобы использовать выходной параметр в Transact-SQL, Вы должны определить ключевое слово OUTPUT в операторах CREATE PROCEDURE и EXECUTE. Если ключевое слово OUTPUT опущено при вызове хранимой процедуры, то она выполняет все вычисления, но не возвращает измененное значение. В большинстве клиентских языках программирования, таких как Microsoft Visual C#®, по умолчанию параметр является входным, таким образом Вы должны указать назначение параметра в клиенте.

Пример использования выходных параметров

Следующий пример создает хранимую процедуру, которая добавляет новый отдел в таблицу **HumanResources.Department** базы данных **AdventureWorks**.

```
CREATE PROC HumanResources.AddDepartment
    @Name nvarchar(50), @GroupName nvarchar(50),
    @DeptID smallint OUTPUT
AS
INSERT INTO HumanResources.Department (Name, GroupName)
    VALUES (@Name, @GroupName)
SET @DeptID = SCOPE_IDENTITY()
```

Выходной параметр **@DeptID** сохраняет идентификатор новой записи, с помощью вызова функции **SCOPE_IDENTITY** для того, чтобы вызывающее приложение могло немедленно обратиться по автоматически сгенерированному идентификационному номеру.

Следующий пример показывает, как вызывающее приложение может сохранить результаты выполнения хранимой процедуры, используя локальную переменную **@dept**.

```
DECLARE @dept int
EXEC AddDepartment 'Refunds', '', @dept OUTPUT
SELECT @dept
```

Возвращаемые значения

Вы можете также вернуть информацию из хранимой процедуры с помощью оператора **RETURN**. Этот метод более ограниченный, чем использование выходных параметров, потому что возвращается только одно целочисленное значение. Оператор **RETURN** обычно используется для возврата результата состояния или кода ошибки процедуры.

Следующий пример изменяет хранимую процедуру **AddDepartment**, чтобы вернуть результат ее выполнения – успешное или неудачное.

```

ALTER PROC HumanResources.AddDepartment
    @Name nvarchar(50), @GroupName nvarchar(50),
    @DeptID smallint OUTPUT
AS
IF ((@Name = "") OR (@GroupName = ""))
    RETURN -1
INSERT INTO HumanResources.Department (Name, GroupName)
VALUES (@Name, @GroupName)

SET @DeptID = SCOPE_IDENTITY()
RETURN 0

```

Если в процедуру передается пустая строка или для **@Name** или для **@GroupName**, то возвращается значение параметра **-1**, чтобы указать ошибку. Если оператор INSERT успешно выполняется, то возвращается **0**.

Следующий пример показывает, как вызывающее приложение может сохранить результат выполнения хранимой процедуры при использовании локальной переменной **@result**.

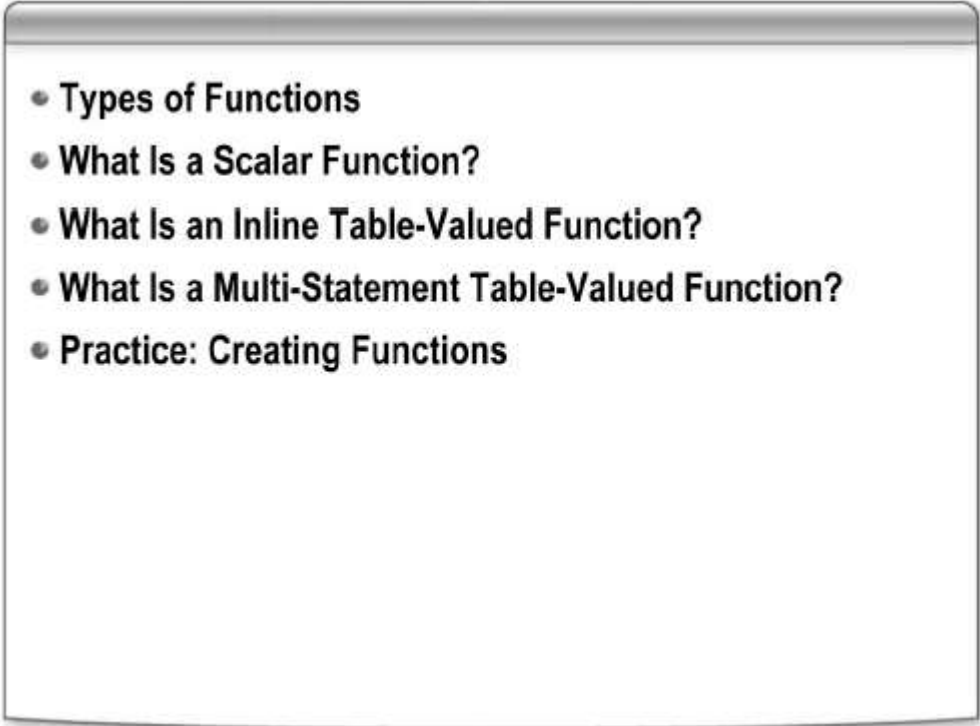
```

DECLARE @dept int, @result int
EXEC @result = AddDepartment 'Refunds', '', @dept OUTPUT
IF (@result = 0)
    SELECT @dept
ELSE
    SELECT 'Error during insert'

```

Замечание. SQL Server автоматически возвращает **0** из хранимых процедур, если Вы не определяете свое собственное значение RETURN.

Урок 3: Создание функций

- 
- **Types of Functions**
 - **What Is a Scalar Function?**
 - **What Is an Inline Table-Valued Function?**
 - **What Is a Multi-Statement Table-Valued Function?**
 - **Practice: Creating Functions**

Цели урока

После завершения этого урока, студенты смогут:

- Описать различные типы функций.
- Описать, как работает скалярная функция.
- Описать, как работает подставляемая табличная функция.
- Описать, как работает многооператорная табличная функция.

Введение

Функции – подпрограммы, которые используются, чтобы формировать часто выполняемую логику. Чем повторять всю эту логику, лучше просто вызывать функцию.

Этот урок предоставляет краткий обзор функций и объясняет, как нужно их использовать, а также рассматривает синтаксис их создания.

Типы функций

- **Scalar functions**
 - Similar to a built-in function
 - Return a single value
- **Inline table-valued functions**
 - Similar to a view with parameters
 - Return a table as the result of single SELECT statement
- **Multi-statement table-valued functions**
 - Similar to a stored procedure
 - Return a new table as the result of INSERT statements

Что такое функция?

Функции – подпрограммы, состоящие из одного или более операторов Transact-SQL, которые формируют код для дальнейшего многократного использования. Функция может иметь *входные параметры* и возвращать *скалярное значение* или *таблицу*. Входные параметры могут быть любого типа данных кроме **timestamp**, **cursor** или **table**, и функции не поддерживают выходные параметры.

Скалярные функции (scalar function)

Скалярные функции возвращают одно значение типа данных, определенного в выражении RETURNS. Эти типы функций синтаксически очень похожи на встроенные системные функции, такие как COUNT() или MAX().

Подставляемые табличные функции (inline table-valued function)

Подставляемая табличная функция возвращает таблицу, которая является результатом одного оператора SELECT. Она подобна представлению, но предлагает больше гибкости, чем представление, потому что для функции можно использовать входные параметры.

Многооператорные табличные функции (multi-statement table-valued function)

Многооператорная табличная функция возвращает таблицу, построенную с помощью одного или более операторов Transact-SQL и подобны хранимой процедуре. В отличие от хранимой процедуры, на мультиоператорную табличную функцию можно сослаться в выражении FROM оператора SELECT, как на представление или таблицу.

Что такое скалярная функция?

- RETURNS clause specifies data type
- Function is defined within a BEGIN...END block

```
CREATE FUNCTION Sales.SumSold(@ProductID int) RETURNS int
AS
BEGIN
    DECLARE @ret int
    SELECT @ret = SUM(OrderQty)
    FROM Sales.SalesOrderDetail WHERE ProductID = @ProductID
    IF (@ret IS NULL)
        SET @ret = 0
    RETURN @ret
END
```

- Can be invoked anywhere a scalar expression of the same data type is allowed

```
SELECT ProductID, Name, Sales.SumSold(ProductID) AS SumSold
FROM Production.Product
```

Создание скалярной функции

Скалярная функция возвращает одно значение типа данных, определенного в выражении RETURNS. Тело функции, определенной в блоке BEGIN ... END, содержит серию операторов Transact-SQL, которые возвращают значение.

Следующий пример создает скалярную функцию, которая подсчитывает общее количество всех продаж для определенного продукта в базе данных **AdventureWorks** и возвращает общее количество как **int**.

```
CREATE FUNCTION Sales.SumSold(@ProductID int) RETURNS int
AS
BEGIN
    DECLARE @ret int
    SELECT @ret = SUM(OrderQty)
    FROM Sales.SalesOrderDetail WHERE ProductID = @ProductID
    IF (@ret IS NULL)
```

SET @ret = 0

RETURN @ret

END

Замечание. При изменении или удалении функции используется синтаксис, подобный изменению или удалению других объектов базы данных. Используйте ALTER FUNCTION, чтобы изменить функцию, и DROP FUNCTION, чтобы удалить функцию из базы данных.

Вызов скалярных функций

Определяемая пользователем функция, которая возвращает скалярное значение, может быть вызвана в операторах Transact-SQL везде, где допустимо скалярное выражение этого типа данных. Следующая таблица содержит примеры использования скалярных функций.

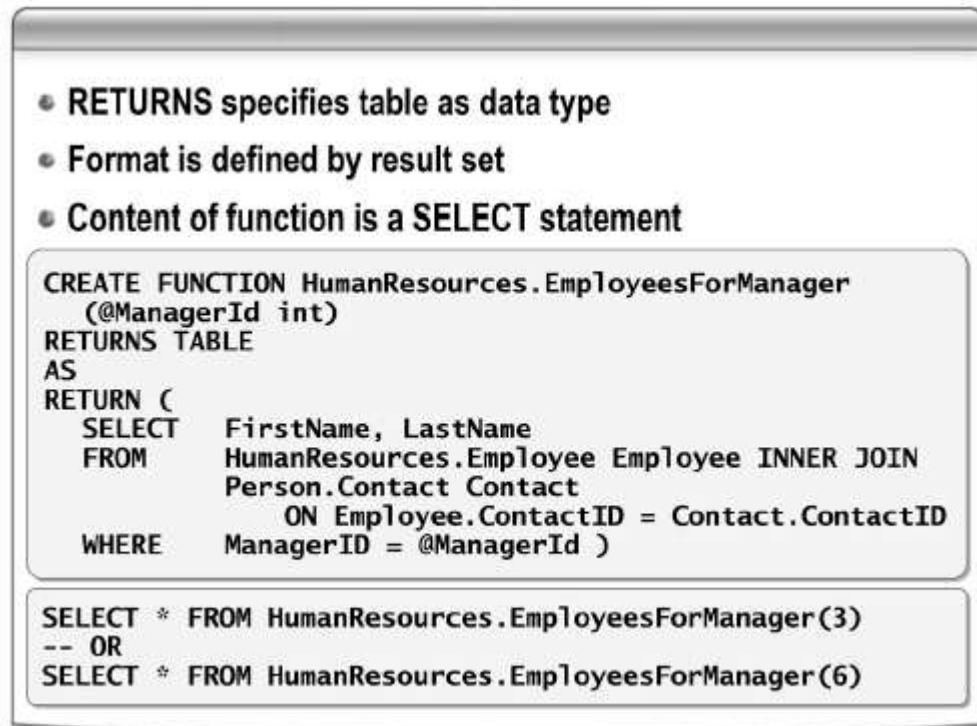
Области	Пример
<u>Запросы</u>	<ul style="list-style-type: none"> ■ Как <i>выражение</i> в <i>select_list</i> оператора SELECT ■ Как <i>выражение</i> или <i>string_expression</i> в выражениях WHERE или HAVING ■ Как <i>group_by_expression</i> в выражении GROUP BY ■ Как <i>order_by_expression</i> в выражении ORDER BY ■ Как <i>expression</i> в выражении SET оператора UPDATE ■ Как <i>expression</i> в выражении VALUES оператора INSERT
<u>Определение таблицы</u>	<ul style="list-style-type: none"> ■ Ограничения CHECK. Функции могут ссылаться только на столбцы той же самой таблицы ■ Определения DEFAULT. Функции могут содержать только константы ■ Вычисляемые столбцы. Функции могут ссылаться только на столбцы одной и той же таблицы
<u>Transact- SQL операторы</u>	<ul style="list-style-type: none"> ■ В операторах присваивания ■ В булевых выражениях операторов управления потоком ■ В выражениях CASE ■ В операторах PRINT (только для функций, возвращающих строку символов)

<u>Функции и хранимые процедуры</u>	<ul style="list-style-type: none"> ■ Как параметры функции ■ Как оператор хранимой процедуры RETURN (только для скалярных функций, которые возвращают целое число). ■ Как оператор RETURN определяемой пользователем функции, предоставляющий значение, которое может быть неявно преобразовано к типу данных вызывающей функции.
-------------------------------------	--

Следующий пример выполняет оператор SELECT, который возвращает поля **ProductID**, **Name**, и результат скалярной функции **SumSold** для каждого продукта, записанного в базе данных **AdventureWorks**.

```
SELECT ProductID, Name, Sales.SumSold(ProductID) AS SumSold
FROM Production.Product
```


Что такое подставляемая табличная функция?



Когда использовать подставляемую табличную функцию

Вы можете использовать подставляемые функции, чтобы достигнуть функциональных возможностей параметризованных представлений. Одним из ограничений представления является то, что нельзя включить параметр в представление при его создании. Конечно, Вы можете сделать это с помощью выражения WHERE уже при вызове представления. Однако, это может потребовать создания строки для динамического выполнения, что может увеличить сложность приложения. Вы можете достигнуть функциональности параметризованного представления с помощью подставляемой табличной функции.

Рассмотрите следующие характеристики подставляемых функций, определяемых пользователем:

- Оператор RETURNS определяет **table** в качестве возвращаемого типа данных.
- Результирующий набор оператора SELECT определяет формат возвращаемой переменной.
- Выражение RETURN содержит один оператор SELECT в круглых скобках. Оператор SELECT, используемый в подставляемой функции, подчиняется тем же самым ограничениям операторов SELECT, используемых в представлениях.
- Тело функции не включается в блок BEGIN ... END.

Пример подставляемой табличной функции

Следующий пример создает подставляемую табличную функцию, которая возвращает имена подчиненных для определенного менеджера в базе данных **AdventureWorks**.

```
CREATE FUNCTION HumanResources.EmployeesForManager
    (@ManagerId int)
RETURNS TABLE
AS
RETURN (
    SELECT FirstName, LastName
    FROM HumanResources.Employee Employee INNER JOIN
        Person.Contact Contact
    ON Employee.ContactID = Contact.ContactID
    WHERE ManagerID = @ManagerId )
```

Вызов подставляемой табличной функции

Используйте подставляемую табличную функцию везде, где Вы обычно использовали бы представление, такое как в выражении FROM оператора SELECT. Следующие примеры позволяют получить имена всех служащих для двух менеджеров.

```
SELECT * FROM HumanResources.EmployeesForManager(3)
-- OR
SELECT * FROM HumanResources.EmployeesForManager(6)
```

Что такое многооператорная табличная функция?

- RETURNS specifies table data type and defines structure
- BEGIN and END enclose multiple statements

```
CREATE FUNCTION HumanResources.EmployeeNames
(@format nvarchar(9))
RETURNS @tbl_Employees TABLE
(EmployeeID int PRIMARY KEY, [Employee Name] nvarchar(100))
AS
BEGIN
    IF (@format = 'SHORTNAME')
        INSERT @tbl_Employees
        SELECT EmployeeID, LastName FROM HumanResources.vEmployee
    ELSE IF (@format = 'LONGNAME')
        INSERT @tbl_Employees
        SELECT EmployeeID, (FirstName + ' ' + LastName)
        FROM HumanResources.vEmployee
    RETURN
END
```

```
SELECT * FROM HumanResources.EmployeeNames('LONGNAME')
```

Когда использовать мультиоператорную табличную функцию

Многооператорная табличная функция – комбинация представления и хранимой процедуры. Вы можете использовать определяемые пользователем функции, которые возвращают таблицу, чтобы заменить хранимую процедуру или представление.

Табличная функция, так же как и хранимая процедура, может использовать сложную логику и множество операторов Transact-SQL, чтобы построить таблицу. Так же как и представление, можно использовать табличную функцию в выражении FROM оператора Transact-SQL.

Рассмотрите следующие характеристики многооператорных табличных функций:

- оператор RETURNS определяет table в качестве возвращаемого типа данных и определяет формат и название для таблицы.
- блок BEGIN ... END ограничивает тело функции.

Пример мультиоператорной табличной функции

Следующий пример создает табличную переменную с двумя столбцами, названную **@tbl_Employees**. Второй столбец изменяется в зависимости от требуемого значения параметра **@format**.

```
CREATE FUNCTION HumanResources.EmployeeNames
    (@format nvarchar(9))
RETURNS @tbl_Employees TABLE
    (EmployeeID int PRIMARY KEY, [Employee Name] nvarchar(100))
AS
BEGIN
    IF (@format = 'SHORTNAME')
        INSERT @tbl_Employees
        SELECT EmployeeID, LastName
        FROM HumanResources.vEmployee
    ELSE IF (@format = 'LONGNAME')
        INSERT @tbl_Employees
        SELECT EmployeeID, (FirstName + ' ' + LastName)
        FROM HumanResources.vEmployee
    RETURN
END
```

Вызов мультиоператорной табличной функции

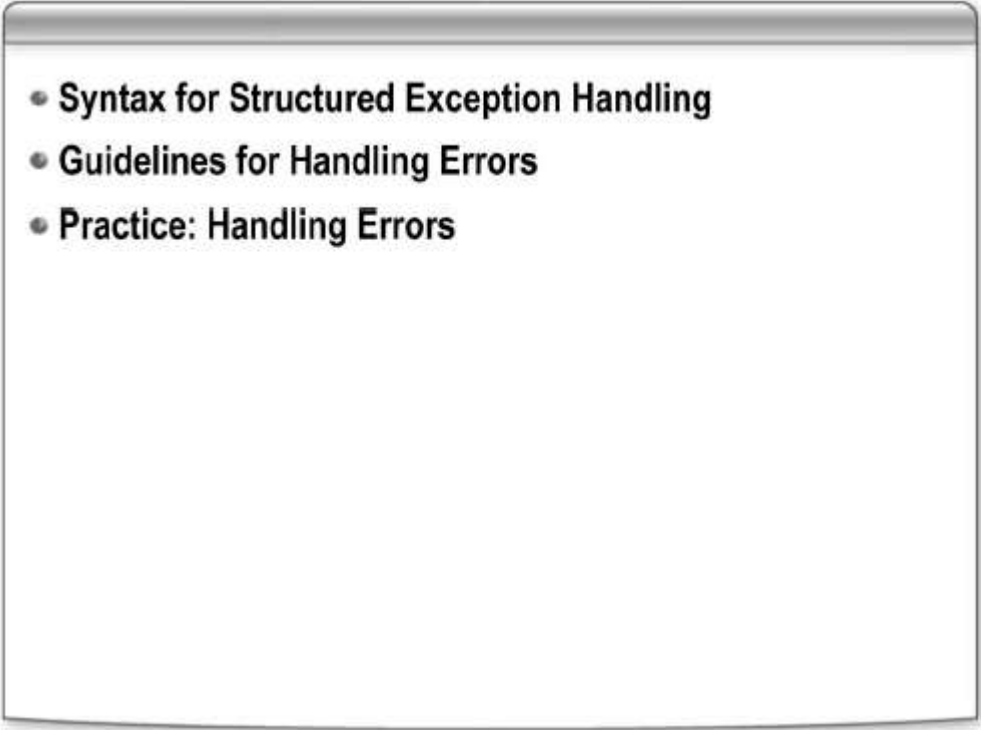
Вы можете вызвать функцию в выражении FROM вместо таблицы или представления. Следующие примеры позволяют получить имя служащего в длинном или коротком формате.

```
SELECT * FROM HumanResources.EmployeeNames('LONGNAME')
```

```
-- OR
```

```
SELECT * FROM HumanResources.EmployeeNames('SHORTNAME')
```

Урок 4: Обработка ошибок

- 
- **Syntax for Structured Exception Handling**
 - **Guidelines for Handling Errors**
 - **Practice: Handling Errors**

Цели урока

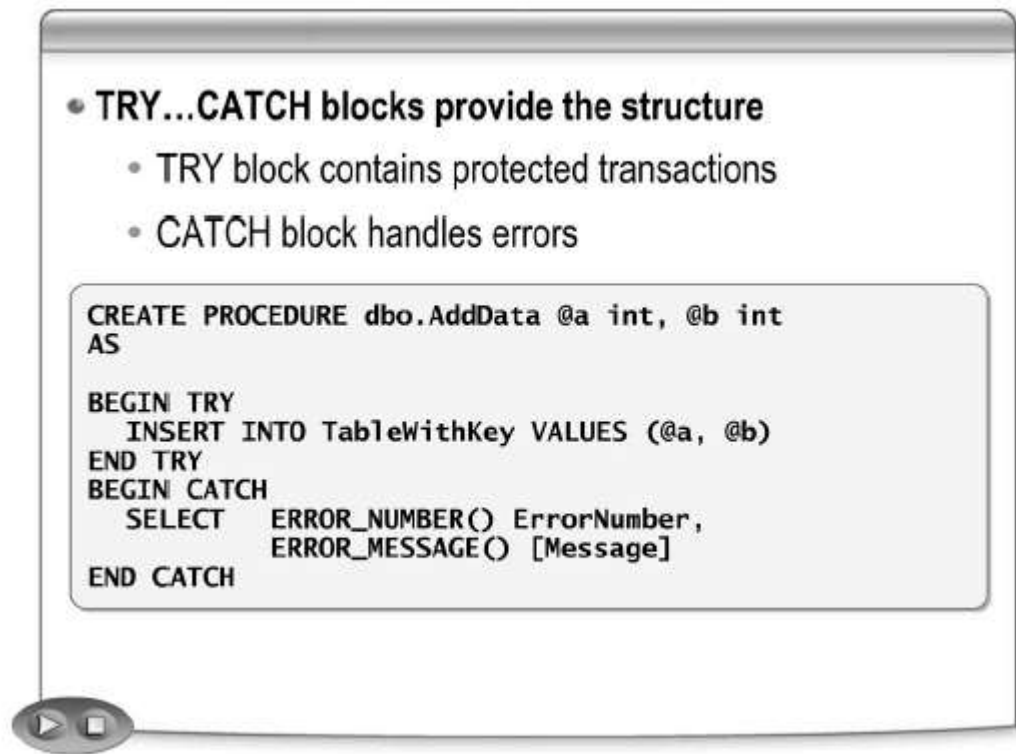
После завершения этого урока, студенты смогут:

- Описать синтаксис для структурной обработки особых ситуаций.
- Описать лучшие методы для ошибок из-за неправильного обращения.

Введение

Этот урок обеспечивает введение в методику структурированной обработки ошибок или исключений, поддерживаемую Microsoft SQL Server 2005. Обработка исключений – важное требование для многих команд Transact-SQL, особенно тех, которые используют транзакции. Структурированная обработка исключений уменьшает объем работы на отладку и делает код более надежным.

Синтаксис для структурированной обработки исключений



Введение

Структурированная обработка исключений является общим способом обработать исключения во многих популярных языках программирования, таких как Microsoft Visual Basic® и Visual C#. SQL Server 2005 позволяет использовать структурированную обработку исключений в любой транзакции, например, такой как хранимая процедура. Это делает код более читаемым и более удобным в сопровождении.

Синтаксис для структурированной обработки ошибок

Чтобы реализовать структурированную обработку исключений, необходимо использовать блоки TRY...CATCH. Блок TRY содержит код транзакции, который потенциально может содержать ошибки. Блок CATCH содержит код, который выполняется, если происходит ошибка в блоке TRY.

У блока TRY...CATCH следующий синтаксис.

BEGIN TRY

{ sql_statement | statement_block }

END TRY

```
BEGIN CATCH
    { sql_statement | statement_block }
END CATCH
```

Здесь *sql_statement* или *statement_block* – любой оператор или группа операторов Transact-SQL.

Пример применения TRY...CATCH

В этом примере хранимая процедура **AddData** пытается вставить два значения в таблицу **TestData**. Первый столбец таблицы **TestData** – целочисленный первичный ключ, а второй столбец имеет целочисленный тип данных. Блок TRY...CATCH в хранимой процедуре **AddData** защищает оператор INSERT для таблицы **TestData** и возвращает номер и сообщение об ошибке в части логики блока CATCH, использующей функции ERROR_NUMBER и ERROR_MESSAGE.

```
CREATE TABLE dbo.TableWithKey (ColA int PRIMARY KEY, ColB int)
GO

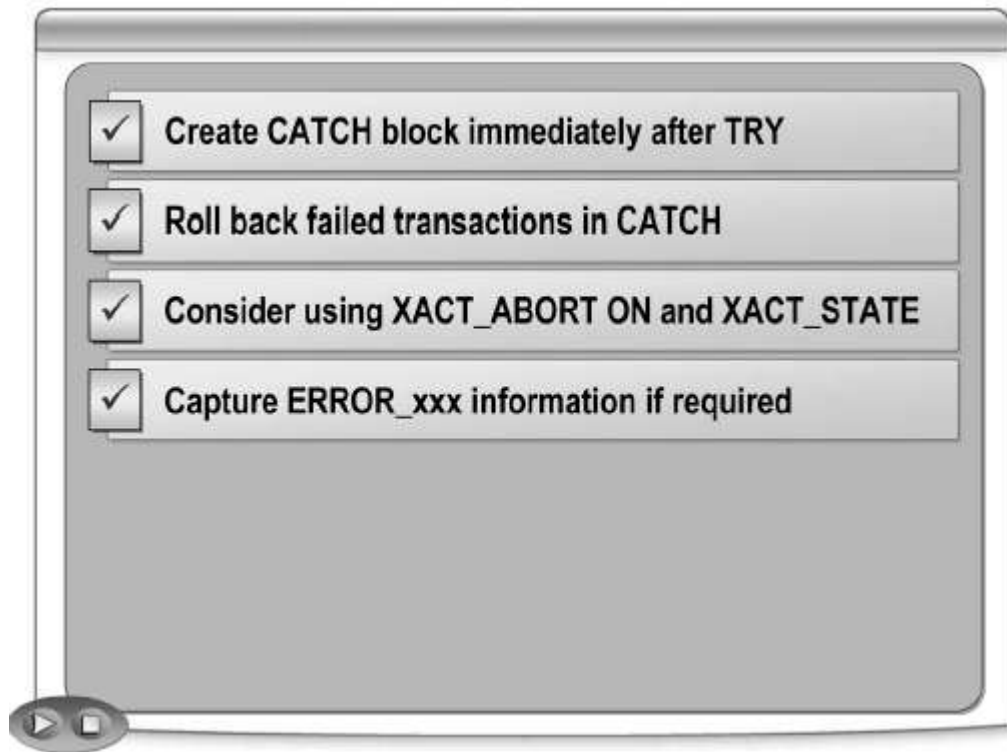
CREATE PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    INSERT INTO TableWithKey VALUES (@a, @b)
END TRY

BEGIN CATCH
    SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH

GO

EXEC dbo.AddData 1, 1
EXEC dbo.AddData 2, 2
EXEC dbo.AddData 1, 3 --violates the primary key
```

Рекомендации по обработке ошибок



Создание блока CATCH

Необходимо создавать блок CATCH сразу же после оператора END TRY при использовании операторов BEGIN CATCH и END CATCH. Нельзя включать какие-либо другие операторы между END TRY и BEGIN CATCH.

Следующий пример выдаст ошибку.

```
BEGIN TRY
```

```
    -- INSERT INTO ...
```

```
END TRY
```

```
    SELECT * FROM TableWithKey -- NOT ALLOWED
```

```
BEGIN CATCH
```

```
    -- SELECT ERROR_NUMBER()
```

```
END CATCH
```


Откат транзакций

Использование транзакций позволяет группировать множество операторов так, чтобы они все завершались успешно или ни один из них не выполнялся бы. Рассмотрим следующий пример, который не использует транзакции.

```
CREATE TABLE dbo.TableNoKey (ColA int, ColB int)
CREATE TABLE dbo.TableWithKey (ColA int PRIMARY KEY, ColB int)
GO

CREATE PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    INSERT dbo.TableNoKey VALUES (@a, @b)
    INSERT dbo.TableWithKey VALUES (@a, @b)
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH
GO

EXEC dbo.AddData 1, 1
EXEC dbo.AddData 2, 2
EXEC dbo.AddData 1, 3          --violates the primary key
```

Этот пример выполняет две последовательных вставки в две различные таблицы. Первая вставка будет всегда успешной, потому что нет ограничения первичного ключа на таблицу. Вторая вставка будет терпеть неудачу всякий раз, когда добавляется повторяющееся значение **ColA**. Поскольку транзакции не используются в этом примере, первая вставка выполнится всегда, даже когда вторая вставка терпит неудачу, то потенциально это может привести к неожиданным результатам.

Следующий пример использует транзакции, чтобы гарантировать, что никакая вставка не выполнится, если хотя бы одна вызовет сбой. Это реализуется при использовании блока **BEGIN TRAN** и **COMMIT TRAN** внутри блока **TRY**, а также оператора **ROLLBACK TRAN** внутри блока **CATCH**.

```

ALTER PROCEDURE dbo.AddData @a int, @b int
AS
BEGIN TRY
    BEGIN TRAN
    INSERT dbo.TableNoKey VALUES (@a, @b)
    INSERT dbo.TablewithKey VALUES (@a, @b)
    COMMIT TRAN
END TRY
BEGIN CATCH
    ROLLBACK TRAN
    SELECT ERROR_NUMBER() ErrorNumber, ERROR_MESSAGE() [Message]
END CATCH
GO

```

Использование опции XACT_ABORT и функции XACT_STATE

Опция XACT_ABORT определяет, откатывает ли SQL Server автоматически до прежнего уровня текущую транзакцию, когда оператор Transact-SQL поднимает ошибку во время выполнения программы. Однако если ошибка происходит в пределах блока TRY, транзакция автоматически не откатывается; вместо этого она становится незавершенной.

Код внутри блока CATCH должен проверить статус транзакции с использованием функции XACT_STATE. XACT_STATE возвращает **-1**, если невыполняемая транзакция присутствует в текущем сеансе. Блок CATCH не должен пытаться завершить транзакцию, а должен откатить ее вручную. Если XACT_STATE возвращает значение **1**, то это означает, что есть транзакция, которая может быть безопасно завершена. Возвращаемое значение **0** означает, что нет никакой текущей транзакции.

Следующий пример устанавливает опцию XACT_ABORT в ON и проверяет статус транзакции внутри блока CATCH.

```

SET XACT_ABORT ON
BEGIN TRY
    BEGIN TRAN
    ...
    COMMIT TRAN
END TRY
BEGIN CATCH
    IF (XACT_STATE()) = -1          -- uncommittable
        ROLLBACK TRAN
    ELSE IF (XACT_STATE()) = 1      -- committable
        COMMIT TRAN
END CATCH

```

Дополнительная информация. Для получения дополнительной информации об использовании XACT_ABORT и XACT_STATE, см. “Использование TRY... CATCH в Transact-SQL” в SQL Server Books Online.

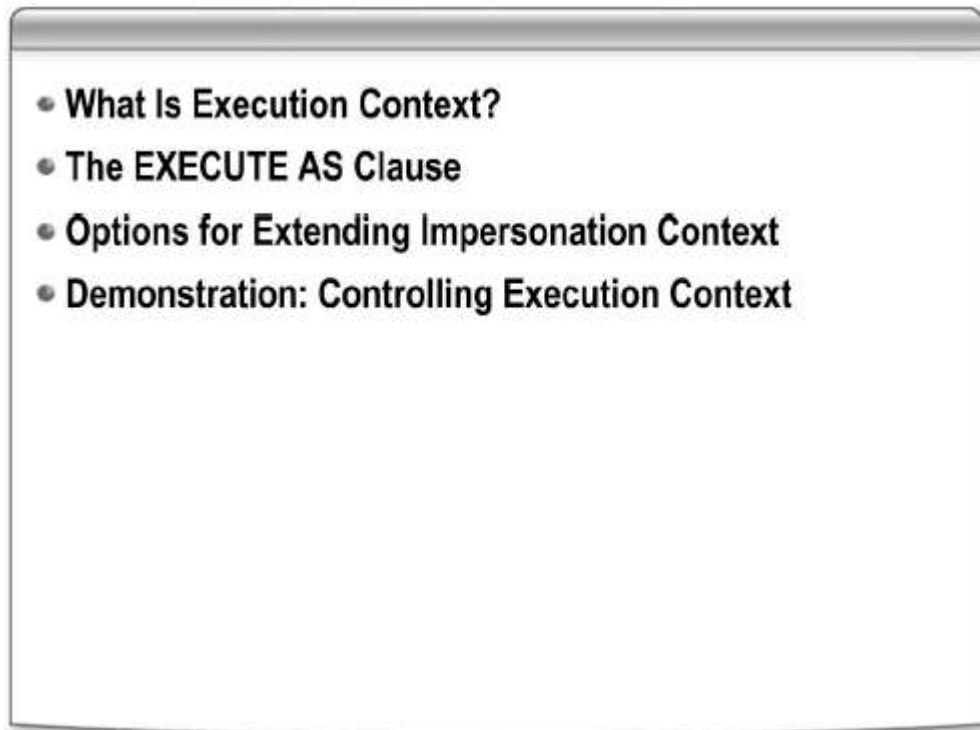
Фиксация информации об ошибках

SQL Server обеспечивает несколько функций, связанных с ошибками, которые можно вызывать внутри блока CATCH для регистрации сведений об ошибках. Например, можно вызвать эти методы и сохранить результаты в таблице файла регистрации ошибок. Следующая таблица перечисляет такие функции.

Функция	Описание
ERROR_LINE	Возвращает номер строки, в которой произошла ошибка, вызвавшая код блока CATCH
ERROR_MESSAGE	Возвращает информацию диагностики о причине ошибки. У многих сообщений об ошибках есть переменные подстановки, в которых помещается такая информация, как название объекта, генерирующего ошибку.
ERROR_NUMBER	Возвращает уникальный номер ошибки.
ERROR_PROCEDURE	Возвращает имя хранимой процедуры или триггера, где произошла ошибка.
ERROR_SEVERITY	Возвращает значение, указывающее, насколько критична ошибка. Ошибки с низкой критичностью, такой как 1 или 2, являются информационными сообщениями или предупреждениями нижнего уровня. Ошибки с высокой критичностью указывают на проблемы, которые нужно решать как можно скорее.
ERROR_STATE	Возвращает значение статуса. Некоторые сообщения об ошибках могут быть вызваны в различных точках кода для Двигателя БД. Каждое определенное условие, которое вызывает ошибку, назначает уникальный статус кода. Эта информация может быть полезной, когда Вы работаете со статьями Knowledge Base Microsoft, чтобы определить, не является ли зарегистрированная проблема той же самой как ошибкой, с которой Вы столкнулись.

Дополнительная информация. Для получения дополнительной информации об уровнях серьезности ошибок, см. “Database Engine Error Severities” в SQL Server Books Online.

Урок 5: Управление контекстом выполнения



Цели урока

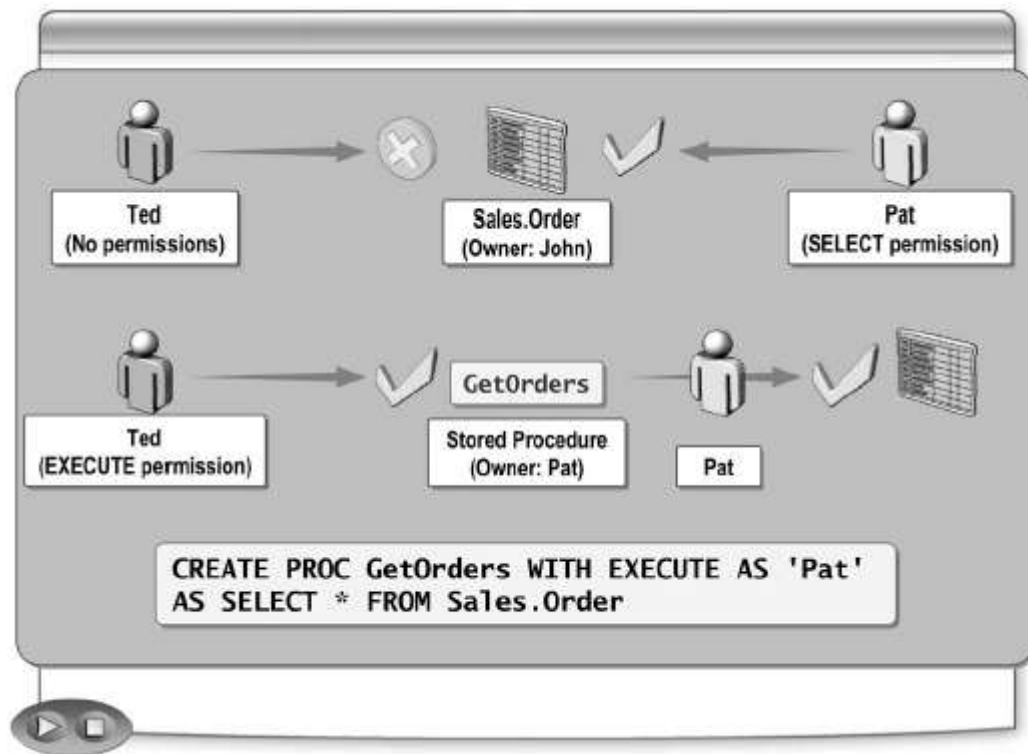
После завершения этого урока, студенты смогут:

- Определять контекст выполнения.
- Описывать, как управлять контекстом выполнения при использовании выражения EXECUTE AS.
- Описывать, как управлять контекстом олицетворения (impersonation context), подписывая (signing) модуль кода.
- Управлять контекстом выполнения.

Введение

В этом уроке Вы узнаете о контексте выполнения и о том, как он влияет на выполнение хранимых процедур и функций. Вы также узнаете, как изменить контекст выполнения при использовании выражения EXECUTE AS и о проблемах, касающихся перекрестного олицетворения в базах данных (cross-database impersonation).

Что такое контекст выполнения?



Определение контекста выполнения

Контекст выполнения устанавливает идентичность (или пользователя), для которой проверяются разрешения. Обычно пользователь или логин, вызывающий программный модуль (хранимую процедуру или функцию) определяет контекст выполнения.

Пример контекста выполнения

Контекст выполнения — идентичность, используемая кодом во время выполнения, и по умолчанию, он означает того, кто вызвал этот код (caller). Однако, в данном случае могут возникнуть проблемы из-за нарушения так называемой *цепочки владения*, как описывается в следующем примере.

Если пользователю по имени **John** принадлежит таблица **Sales.Order** и он предоставляет разрешение **SELECT** только пользователю **Pat**, то пользователь **Tad** не может обратиться к этой таблице, как показано на слайде.

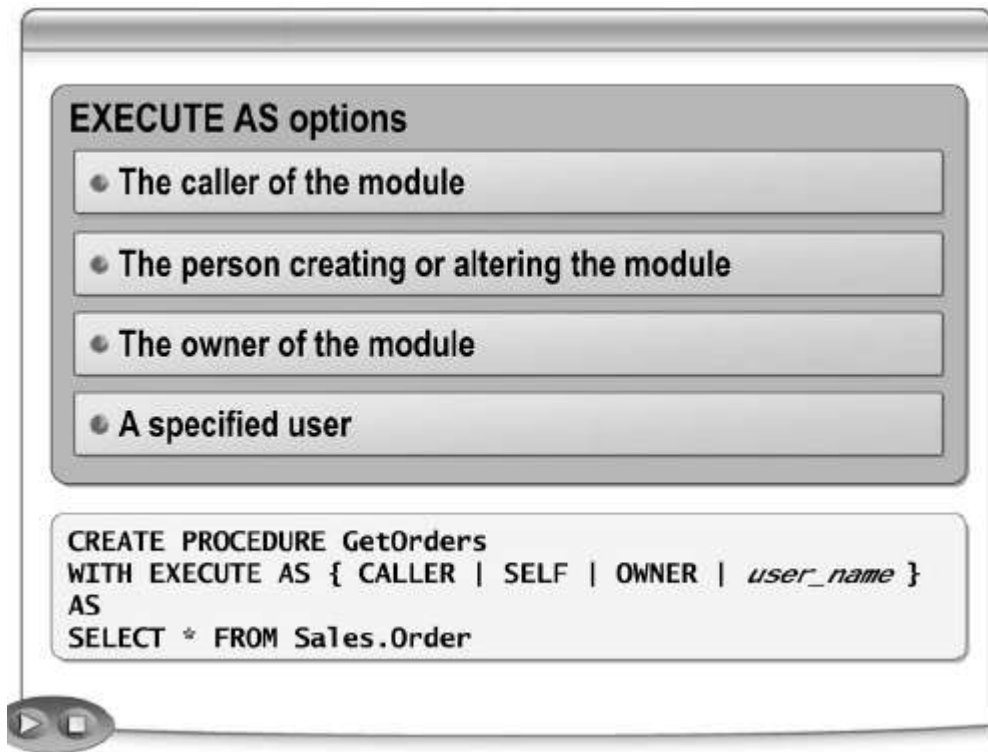
Если пользователю **Pat** принадлежит хранимая процедура **GetOrders**, которая читает данные из таблицы **Sales.Order**, то **Pat** может выполнить эту хранимую процедуру без каких-либо проблем безопасности, потому что **Pat** имеет разрешение **SELECT** на данную таблицу.

Если **Pat** дает разрешение **EXECUTE** на хранимую процедуру пользователю **Tad**, это привело бы к ошибке в случае, когда **Tad** попытался бы выполнить процедуру. Это происходит потому, что по умолчанию хранимая процедура выполняется от имени пользователя **Tad**, а у него нет разрешения **SELECT** на таблицу **Sales.Order**.

Чтобы позволить пользователю **Tad** успешно выполнять хранимую процедуру, контекст выполнения нужно изменить на пользователя, который имеет соответствующие разрешения. Нужно использовать выражение **EXECUTE AS**, чтобы переключить контекст выполнения для хранимой процедуры, как описывается в следующей теме.

Замечание. Этот пример также демонстрирует нарушение цепочки владения, потому что таблица **Sales.Order** и хранимая процедура **GetOrders** принадлежат различным пользователям. Для получения дополнительной информации о цепочках владения, см. “Ownership Chains” в SQL Server Books Online.

Выражение EXECUTE AS



Введение

Вы можете использовать выражение EXECUTE AS в хранимой процедуре или функции, чтобы указать идентичность, используемую в его контексте выполнения. Понимание того, как использовать выражение EXECUTE AS может помочь реализовать безопасность в сценариях, в которых необходимо обращаться к зависимым объектам, но Вы не хотите нарушать цепочки владения.

Опции EXECUTE AS

Вы можете использовать выражение EXECUTE AS в командах CREATE PROCEDURE и CREATE FUNCTION, кроме объявлений подставляемых табличных функций (inline table-valued functions). Здесь показан синтаксис для выражения EXECUTE AS.

```
EXECUTE AS { CALLER | SELF | OWNER | user_name }
```


Опции, включенные в синтаксис выражения EXECUTE AS, описываются в следующей таблице.

Опции	Описание
CALLER	Выполняется при использовании идентичности вызывающего пользователя. Это установка действует по умолчанию.
SELF	Выполняется при использовании идентичности пользователя, который создает или изменяет хранимую процедуру или функцию. Это значение не меняется, даже если другой пользователь становится владельцем модуля.
OWNER	Выполняется при использовании идентичности владельца функции. Это значение меняется, если другой пользователь становится владельцем модуля.
<i>user_name</i>	Выполняются при использовании идентичности указанного пользователя. Если <i>user_name</i> имеет то же значение, что и пользователь, создавший или изменивший модуль, то выражение EXECUTE AS <i>user_name</i> эквивалентно EXECUTE AS SELF.

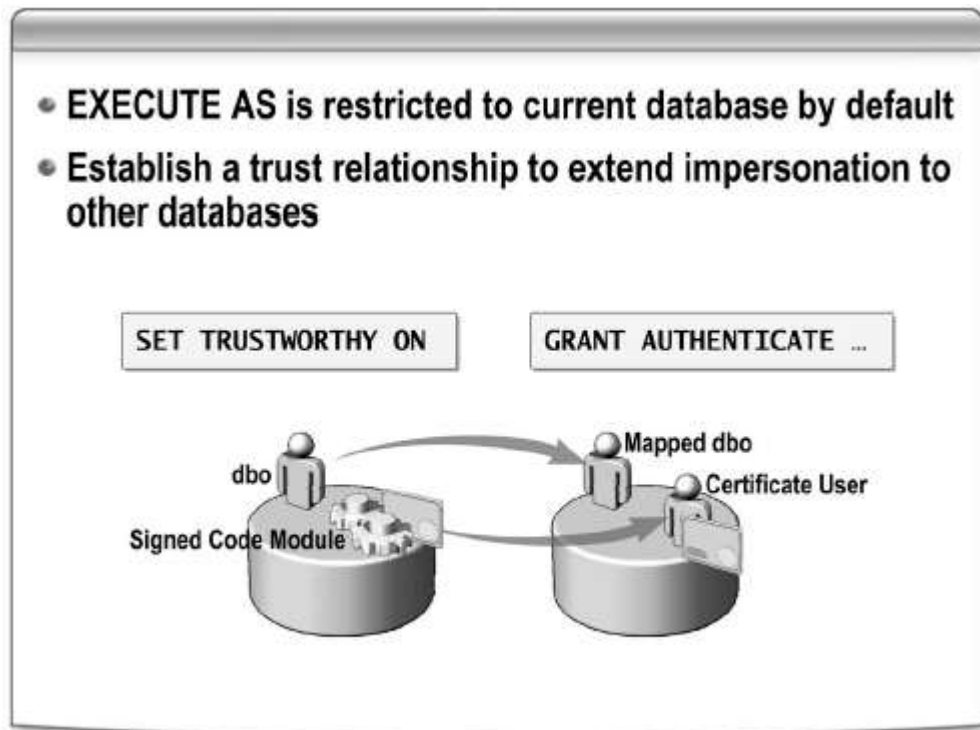
Пример использования EXECUTE AS

Чтобы решить проблему, описанную в предыдущей теме, необходимо определить в качестве *user_name* **Pat**, что позволит пользователю **Tad** выполнять успешно хранимую процедуру, как показано в следующем примере.

```
CREATE PROCEDURE GetOrders
WITH EXECUTE AS 'Pat'
AS
SELECT * FROM Sales.Order
```

Замечание. Вы можете также использовать EXECUTE AS как автономный оператор Transact-SQL для временного изменения текущего контекста выполнения для любого подключения до тех пор, пока не выполнен оператор REVERT. Для получения дополнительной информации см., "REVERT (Transact-SQL)" в SQL Server Books Online.

Опции для расширения контекста олицетворения



Ограничения EXECUTE AS

Когда Вы используете выражение `EXECUTE AS`, чтобы изменить контекст выполнения так, чтобы код модуля выполнялся бы от имени указанного пользователя, то в этом случае код, как говорят, "олицетворяет" альтернативного пользователя. По умолчанию получающийся контекст олицетворения будет работать только в пределах области видимости текущей базы данных. Это означает что, если Вы создаете хранимую процедуру, которая использует таблицу в другой базе данных, выражение `EXECUTE AS` передаст контекст олицетворения другой базе данных, но данный контекст будет недопустим.

Установление доверительных отношений для расширения олицетворения

Вы можете выборочно расширить область видимости олицетворения на базу данных, установленную в пределах другой базы данных, установив трастовую модель между этими двумя базами данных. Это было бы полезно для приложения, которое использует две базы данных и требует доступа к одной базе данных из другой база данных.

SQL Server 2005 использует аутентификаторы, чтобы определить, является ли установленный контекст правильным в пределах специфической области видимости. Часто аутентификатор – это либо любой системный администратор, либо экземпляр SQL Server, либо пользователь

dbo в указанных базах данных. Аутентификатор – владелец области видимости, в пределах которой установлены контекст для специфического пользователя или логина.

Валидность контекста олицетворения пользователя вне базы данных, где этот контекст установлен, зависит от того, доверяют ли аутентификатору данного контекста в целевой области видимости.

Это доверие устанавливается созданием двойного логина аутентификатора и предоставления разрешения **AUTHENTICATE** (подтвердить подлинность) в случае, если целевая область видимости – другая база данных или предоставления разрешения **AUTHENTICATE SERVER**, если целевая область видимости – экземпляр SQL Server. Кроме того, запрашивающая база данных должна быть отмечена как **TRUSTWORTHY** (заслуживающая доверия).

Основные шаги установления трасовых отношений для пользователя **dbo**:

1. Создайте пользователя в целевой базе данных с тем же самым логином, что и **dbo** в вызывающей базе данных. Он будет аутентификатором в целевой базе данных.
2. Предоставьте отображенному пользователю разрешение **AUTHENTICATE** (или) в целевой базе данных.
3. Предоставьте аутентификатору разрешения, требуемые хранимой процедуре в вызывающей базе данных.
4. Измените вызывающую базу данных – установите опцию **TRUSTWORTHY** в **ON**.

Использование сертификатов и подписей для валидации вызывающих программ

Альтернативный способ установить трасовые отношения между базами данных состоит в том, чтобы использовать сертификаты или асимметричные ключи как аутентификаторы. Это дает преимущество в использовании методики, называемой подписание.

Подпись на модуле верифицирует то, что человек может изменить код в пределах модуля, только если у этого человека есть доступ к секретному ключу, который используется для подписания модуля. Это позволяет устанавливать трасовые отношения с сертификатом, используемым для подписания скорее, чем только с владельцем базы данных. Доверие подписанного модуля достигается предоставлением пользователю в целевой области видимости разрешения **AUTHENTICATE** или **AUTHENTICATE SERVER**, которые отображаются к сертификату или асимметричному ключу.

Основные шаги для использования сертификата и подписи, чтобы проверить правильность вызывающих программ:

1. Создайте сертификат при использовании оператора **CREATE CERTIFICATE** в вызывающей базе данных.

2. Добавьте подпись к вызывающей хранимой процедуре при использовании `ADD SIGNATURE`.
3. Импортируйте сертификат в целевую базу данных. Это может быть достигнуто копированием (backing up) сертификата из вызывающей базы данных в файл, а затем использованием `CREATE CERTIFICATE` в целевой базе данных со ссылкой на файл-копию сертификата.
4. Создайте пользователя на основе сертификата при использовании выражения `FROM CERTIFICATE` оператора `CREATE USER` и дайте этому пользователю соответствующие разрешения на требуемые объекты базы данных.
5. Дайте этому пользователю разрешение `AUTHENTICATE` или `AUTHENTICATE SERVER`.

Дополнительная информация. Для получения дополнительной информации о расширении контекста олицетворения см. раздел “Extending Database Impersonation by Using `EXECUTE AS`” в SQL Books Online.