COMP 430
Intro. to Database Systems
Encapsulating SQL code
Want to bundle SQL into code blocks

Like in every other language

Encapsulation
Abstraction
Code reuse
Maintenance
...

DB- or application-level?

DB:
- **Encapsulation** – Application programmer only uses API, and doesn’t need details, including much SQL.
- **Safety** – Protect DB from application programmer. Only expose desired API.
- **Performance** – Less I/O. Better DBMS optimization.

Application:
- **Convenience** – More Java/C++/… programmers than for SQL.
- **Flexibility** – Can use the DB however you want.
- **Libraries** – Make everything easier.
Procedural extensions of SQL

- PL/SQL – Oracle, IBM, ...
- PL/pgSQL – PostgreSQL
- Transact-SQL – Microsoft, Sybase

We’ll use this.

Adds variables, assignment, conditionals, loops, begin/end, procedures, functions, packages(?), security, ...

Yet another full language. We’ll focus on only some core concepts.
Security

Don’t trust some users:
Restrict access to tables, rows, columns, procedures, or functions based upon user/group/role
GRANT, DENY, REVOKE, ...

Don’t trust some programmers:
Namespaces – packages/schemas
“Private” functions/procedures

“Private” not supported in Transact-SQL. We can define API, but can’t prevent SQL code from bypassing API.

In this course: Only consider SQL injection attacks (later).
DB-level code blocks
## Three types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined functions</td>
<td>Side-effect-free mappings from inputs to output</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>Primarily used for their side-effects, e.g., updating DB</td>
</tr>
<tr>
<td>Triggers</td>
<td>Event handlers</td>
</tr>
</tbody>
</table>
Scalar function example

CREATE FUNCTION CubeVolume (@side_len INT READONLY) RETURNS INT WITH RETURNS NULL ON NULL INPUT AS BEGIN DECLARE @volume INT; IF @side_len < 0 THEN SET @volume = 0; ELSE SET @volume = @side_len * @side_len * @side_len; RETURN @volume; END;
Inline table-valued function example

CREATE FUNCTION GetEmployeeIDsCountry (@country VARCHAR(50) READONLY)
RETURNS TABLE
AS
RETURN (SELECT id FROM Employee WHERE country = @country);
CREATE FUNCTION ParseDelimitedInts (@list VARCHAR(5000),  @delimiter VARCHAR(10) = ',', READONLY)
RETURNS @int_table TABLE (item INT)
AS
BEGIN
    DECLARE @text_item VARCHAR(200);
    DECLARE @index INT;
    WHILE Datalength(@list) > 0
        BEGIN
            SET @index = Charindex(@delimiter, @list);
            IF @index > 0
                BEGIN
                    SET @text_item = Substring(@list, 1, @index - 1);
                    SET @list = Substring(@list, @index + Datalength(@delimiter), Datalength(@list));
                END
            ELSE
                BEGIN
                    SET @item_text = @list;
                    SET @list = NULL;
                END
            INSERT INTO @int_table
                SELECT item = Convert(INT, @text_item);
        END
    RETURN
END
CREATE PROCEDURE InsertEmployee (@first_name VARCHAR(50),
@last_name VARCHAR(50),
@salary INT,
@hire_date DATETIME)
AS
BEGIN
    DECLARE @count INT;
    SELECT @count = COUNT(*)
    FROM Employee
    WHERE first_name = @first_name
        AND last_name = @last_name;
    IF @count = 0
        INSERT INTO Employee VALUES (@first_name, @last_name, @salary, @hire_date);
END

EXECUTE InsertEmployee 'John', 'Smith', 50000, '03-11-2016';

EXECUTE InsertEmployee @first_name = 'Mary', @last_name = 'Jones',
                      @salary = 60000, @hire_date = '02-12-2016';
Cursors often used in procedures

SELECT **implicitly** loops over table data.

*Cursors* allow SQL code to **explicitly** loop over table data.

- Generally **much** slower than declarative SQL. (Often written by SQL-newbies.)
- Mechanism for returning table from procedure (soon).
- Concept sometimes used at application-level (soon).
+ Useful for running totals.
+ Decreases transaction granularity (later).

**Typical usage (pseudo-code):**

Cursor is for SELECT ...
While cursor not at end of query
Do something with this row.
Move cursor to next row.
CREATE PROCEDURE MyProcedure (...) 
AS 
BEGIN 
    DECLARE @id INT; 
    DECLARE @first_name VARCHAR(50); 
    DECLARE @last_name VARCHAR(50); 

    DECLARE db_cursor CURSOR FOR 
    SELECT id, first_name, last_name FROM Student; 

    OPEN db_cursor; 
    FETCH NEXT FROM db_cursor INTO @id, @first_name, @last_name; 
    WHILE @@FETCH_STATUS = 0 
    BEGIN 
        ... 
        FETCH NEXT FROM db_cursor INTO @id, @first_name, @last_name; 
    END 
    CLOSE db_cursor; 
    DEALLOCATE db_cursor; 
END
Returning values from a procedure

User-defined functions can return scalars or tables. What about procedures?

• Can return scalars
• Can sort-of return tables
• Several different mechanisms
Returning via OUTPUT parameters

```
CREATE PROCEDURE MyProcedure (
    @a VARCHAR(50),
    @b INT OUTPUT)
AS
BEGIN
  ...
END
```

Scalars only. Not text. Not tables.

```
DECLARE @my_a VARCHAR(50) = "foo";
DECLARE @my_b INT = 42;
EXECUTE MyProcedure
    @my_a,
    @my_b OUTPUT;
```

Functions cannot have OUTPUT parameters.
Returning via RETURN

CREATE PROCEDURE MyProcedure ( 
    @a VARCHAR(50), 
    @b INT)
AS
BEGIN
...
    RETURN 2;
END

DECLARE @status INT;
EXECUTE @status = MyProcedure 
    “foo”,
    42;

INT only. Meant as a status code.

Not meant as a general-purpose technique.
Returning result set

CREATE PROCEDURE MyProcedure (  
    @a VARCHAR(50),  
    @b INT)  
AS  
BEGIN  
    SELECT ...;  
END

INSERT INTO table_name  
EXECUTE MyProcedure  
    @my_a,  
    @my_b;

Returns a temporary table of data.
Returning a cursor

CREATE PROCEDURE MyProcedure ( 
    @a VARCHAR(50),
    @b INT,
    @cursor CURSOR VARYING OUTPUT)
AS
BEGIN
    SET @cursor = CURSOR FOR
    SELECT ...;
END

DECLARE @my_cursor CURSOR;
EXECUTE MyProcedure 
    @my_a,
    @my_b,
    @my_cursor OUTPUT;

In effect, returns the table itself.
Triggers

A stored procedure associated with a particular table that is *fired* when an INSERT, UPDATE, or DELETE occurs.

![Code example]

```sql
CREATE TRIGGER TrigAddStudent ON Students AFTER INSERT, UPDATE
AS
BEGIN
    DECLARE @count INT;
    SELECT @count = COUNT(*) in Inserted;
    PRINT @count + ‘ students added or updated.’
END
```

Alternate:

```sql
INSTEAD OF
```

Similarly, Deleted is available for UPDATE, DELETE.

No parameters.
Main uses of triggers

• **Logging**

• Automatic correction of bad data.

• Updating relevant data elsewhere for consistency.
  • E.g., PurchasItem table has trigger that updates Inventory table.

• Bad: Rejecting bad values – use CHECK.