COMP 430
Intro. to Database Systems

Encapsulating SQL code

Want code blocks, as in other languages

Encapsulation
Abstraction
Code reuse
Maintenance
…

Procedural extensions of SQL

- PL/SQL – Oracle, IBM, …
- PL/pgSQL – PostreSQL
- Transact-SQL – Microsoft, Sybase

We'll use this.

Adds variables, assignment, conditionals, loops, begin/end, functions, exceptions, namespaces, security, transactions, …

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

Security at a glance

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

Kinds of code blocks – most SQLs

(User def'd) functions

Side-effect-free

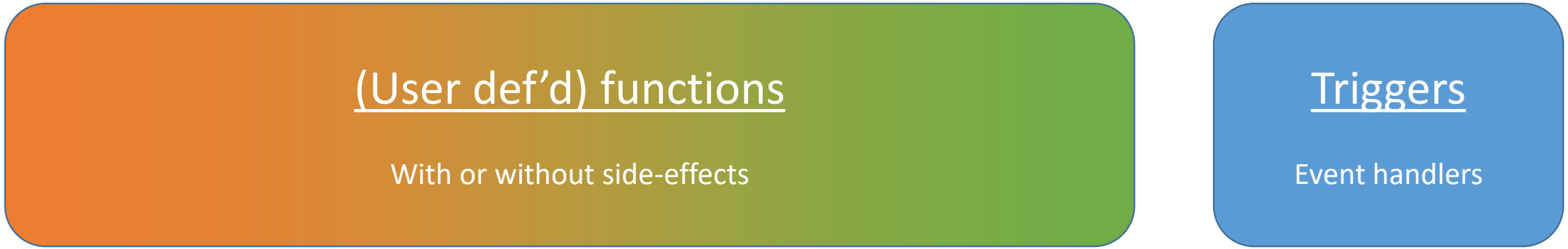
(Stored) procedures

Primarily used for side-effects, e.g., updating DB

Triggers

Event handlers

Kinds of code blocks – PostgreSQL



```
CREATE OR REPLACE FUNCTION CubeVolume (side_len INT)
RETURNS INT
RETURNS NULL ON NULL INPUT
AS $$
    DECLARE
        volume INT;
    BEGIN
        IF side_len < 0
        THEN
            volume := 0;
        ELSE
            volume := side_len * side_len * side_len;
        END IF;
        RETURN volume;
    END $$
LANGUAGE plpgsql
IMMUTABLE;
```

```
SELECT CubeVolume(10);
```

```
CREATE OR REPLACE FUNCTION Factorial (n NUMERIC)
RETURNS NUMERIC
AS $$
   DECLARE
      i NUMERIC;
      result NUMERIC;
   BEGIN
      result := 1;
      FOR i IN 1 .. n LOOP
         result := result * i;
      END LOOP;
      RETURN result;
   END; $$
LANGUAGE plpgsql
IMMUTABLE;
```

```
SELECT Factorial(25::NUMERIC);
```

```
CREATE OR REPLACE FUNCTION Factorial (n NUMERIC)
RETURNS NUMERIC
AS $$
   DECLARE
      i NUMERIC;
      result NUMERIC;
   BEGIN
      i := 1;
      result := 1;
      WHILE i <= n LOOP
         result := result * i;
         i := i + 1;
      END LOOP;
      RETURN result;
   END; $$
LANGUAGE plpgsql
IMMUTABLE;
```

```
SELECT Factorial(25::NUMERIC);
```

```
CREATE OR REPLACE FUNCTION Factorial (n NUMERIC)
RETURNS NUMERIC
AS $$
   BEGIN
      IF n = 0 THEN
         RETURN 1;
      ELSE
         RETURN n * Factorial(n - 1);
      END IF;
   END; $$
LANGUAGE plpgsql
IMMUTABLE;
```

```
SELECT Factorial(25::NUMERIC);
```

```
CREATE FUNCTION GetEmployeeIDsCountry (country_value VARCHAR(50))
RETURNS TABLE (id UUID)
AS $$
   BEGIN
      RETURN QUERY SELECT id FROM Employee WHERE country = country_value;
   END $$
LANGUAGE plpgsql
STABLE;
```

```
SELECT *
FROM GetEmployeeIDsCountry (“Denmark”);
```

```
CREATE FUNCTION InsertEmployee (
   emp_first_name VARCHAR(50),
   emp_last_name VARCHAR (50),
   emp_salary INT,
   emp_hire_date DATETIME)
RETURNS VOID
AS $$
   DECLARE
      matches INT;
   BEGIN
      SELECT matches = COUNT(*)
      FROM Employee
      WHERE first_name = emp_first_name AND
                last_name = emp_last_name;
      IF matches = 0
         INSERT INTO Employee VALUES
                   (emp_first_name, emp_last_name,
                    emp_salary, emp_hire_date);
      END IF;
   END $$
LANGUAGE plpgsql;
```

```
PERFORM InsertEmployee
      ('John', 'Smith', 50000, '03-11-2016');

PERFORM InsertEmployee
      (emp_first_name := 'Mary',
       emp_last_name := 'Jones',
       emp_salary := 60000,
       emp_hire_date := '02-12-2016');
```

```
CREATE OR REPLACE FUNCTION Squares (_tbl REGCLASS)
RETURNS TABLE (n FLOAT, square FLOAT)
AS $$
   BEGIN
      RETURN QUERY EXECUTE ‘SELECT n, n^2 FROM ‘ || _tbl;
   END $$
LANGUAGE plpgsql;
```

```
CREATE TABLE Data (n FLOAT);

…

SELECT * FROM Squares(‘Data’);
```

Triggers

Triggers & main uses

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

- **Logging**
- Automatic correction of bad data.
- Updating relevant data elsewhere for consistency.
  - E.g., PurchaseItem table has trigger that updates Inventory table.
- Rejecting bad values
  - Use CHECK, instead, when sufficient.

Alternative: Provide API with Purchase() function to update the tables.

```
CREATE TABLE Emp (
   empname TEXT,
   salary INT,
   last_date TIMESTAMP,
   last_user TEXT);
```

```
CREATE FUNCTION EmpStamp()
RETURNS TRIGGER
AS $$
   BEGIN
      IF NEW.empname IS NULL THEN
         RAISE EXCEPTION ‘empname cannot be null’;
      ELSIF NEW.salary IS NULL THEN
         RAISE EXCEPTION ‘% cannot have null salary’, NEW.empname;
      ELSIF NEW.salary < 0 THEN
         RAISE EXCEPTION ‘% cannot have a negative salary’, NEW.empname;
      ELSE
         NEW.last_date := current_timestamp;
         NEW.last_user := current_user;
      END IF;
      RETURN NEW;
   END $$
LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON Emp
   FOR EACH ROW EXECUTE PROCEDURE EmpStamp();
```