Basic Trigger Syntax

Below is the syntax for creating a trigger in Oracle (which differs slightly from standard SQL syntax):

```
CREATE [OR REPLACE] TRIGGER <trigger_name>

 {BEFORE|AFTER} {INSERT|DELETE|UPDATE} ON <table_name>

 [REFERENCING [NEW AS <new_row_name>] [OLD AS <old_row_name>]]

 [FOR EACH ROW [WHEN (<trigger_condition>)]]

 <trigger_body>
```

Some important points to note:

- You can create only `BEFORE` and `AFTER` triggers for tables. (`INSTEAD OF` triggers are only available for views; typically they are used to implement view updates.)
- You may specify up to three triggering events using the keyword `OR`. Furthermore, `UPDATE` can be optionally followed by the keyword `OF` and a list of attribute(s) in `<table_name>`. If present, the `OF` clause defines the event to be only an update of the attribute(s) listed after `OF`. Here are some examples:

  ```
  ... INSERT ON R ...

  ... INSERT OR DELETE OR UPDATE ON R ...

  ... UPDATE OF A, B OR INSERT ON R ...
  ```

- If `FOR EACH ROW` option is specified, the trigger is row-level; otherwise, the trigger is statement-level.
- Only for row-level triggers:
  - The special variables `NEW` and `OLD` are available to refer to new and old tuples respectively. **Note:** In the trigger body, `NEW` and `OLD` must be preceded by a colon ("`:`"), but in the `WHEN` clause, they do not have a preceding colon! See example below.
  - The `REFERENCING` clause can be used to assign aliases to the variables `NEW` and `OLD`.
  - A trigger restriction can be specified in the `WHEN` clause, enclosed by parentheses. The trigger restriction is a SQL condition that must be satisfied in order for Oracle to fire the trigger. This condition cannot contain subqueries. Without the `WHEN` clause, the trigger is fired for each row.
- `<trigger_body>` is a PL/SQL block, rather than sequence of SQL statements. Oracle has placed certain restrictions on what you can do in `<trigger_body>`, in order to avoid situations where one trigger performs an action that triggers a second trigger, which then triggers a third, and so on, which could potentially create an infinite loop. The restrictions on `<trigger_body>` include:
  - You cannot modify the same relation whose modification is the event triggering the trigger.
  - You cannot modify a relation connected to the triggering relation by another constraint such as a foreign-key constraint.

Trigger Example

We illustrate Oracle's syntax for creating a trigger through an example based on the following two tables:

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));

CREATE TABLE T5 (c CHAR(10), d INTEGER);
```

We create a trigger that may insert a tuple into `T5` when a tuple is inserted into `T4`. Specifically, the trigger checks whether the new tuple has a first component 10 or less, and if so inserts the reverse tuple into `T5`:

```
CREATE TRIGGER trig1
    AFTER INSERT ON T4
    REFERENCING NEW AS newRow
    FOR EACH ROW
    WHEN (newRow.a <= 10)
    BEGIN
        INSERT INTO T5 VALUES(:newRow.b, :newRow.a);
    END trig1;
.
run;
```

Notice that we end the `CREATE TRIGGER` statement with a dot and `run`, as for all PL/SQL statements in general. Running the `CREATE TRIGGER` statement only creates the trigger; it does not execute the trigger. Only a triggering event, such as an insertion into `T4` in this example, causes the trigger to execute.

Displaying Trigger Definition Errors

As for PL/SQL procedures, if you get a message

```
Warning: Trigger created with compilation errors.
```

you can see the error messages by typing

```
show errors trigger <trigger_name>;
```

Alternatively, you can type, `SHO ERR` (short for `SHOW ERRORS`) to see the most recent compilation error. Note that the reported line numbers where the errors occur are not accurate.

Viewing Defined Triggers

To view a list of all defined triggers, use:

```
select trigger_name from user_triggers;
```

For more details on a particular trigger:

```
select trigger_type, triggering_event, table_name, referencing_names, trigger_body
from user_triggers
where trigger_name = '<trigger_name>';
```

Dropping Triggers

To drop a trigger:

```
drop trigger <trigger_name>;
```

Disabling Triggers

To disable or enable a trigger:

```
alter trigger <trigger_name> {disable|enable};
```

Aborting Triggers with Error

Triggers can often be used to enforce contraints. The WHEN clause or body of the trigger can check for the violation of certain conditions and signal an error accordingly using the Oracle built-in function RAISE_APPLICATION_ERROR. The action that activated the trigger (insert, update, or delete) would be aborted. For example, the following trigger enforces the constraint Person.age >= 0:

```
create table Person (age int);
```

```
CREATE TRIGGER PersonCheckAge
AFTER INSERT OR UPDATE OF age ON Person
FOR EACH ROW
BEGIN
    IF (:new.age < 0) THEN
        RAISE_APPLICATION_ERROR(-20000, 'no negative age allowed');
    END IF;
END;
.
RUN;
```

If we attempted to execute the insertion:

```
insert into Person values (-3);
```

we would get the error message:

```
ERROR at line 1:
ORA-20000: no negative age allowed
ORA-06512: at "MYNAME.PERSONCHECKAGE", line 3
ORA-04088: error during execution of trigger 'MYNAME.PERSONCHECKAGE'
```

and nothing would be inserted. In general, the effects of both the trigger and the triggering statement are rolled back.

Mutating Table Errors

Sometimes you may find that Oracle reports a "mutating table error" when your trigger executes. This happens when the trigger is querying or modifying a "mutating table", which is either the table whose modification activated the trigger, or a table that might need to be updated because of a foreign key constraint with a CASCADE policy. To avoid mutating table errors:

- A row-level trigger must not query or modify a mutating table. (Of course, NEW and OLD still can be accessed by the trigger.)
- A statement-level trigger must not query or modify a mutating table if the trigger is fired as the result of a CASCADE delete.