Exercises for Chapter 13: Triggers

The Labs below provide you with exercises and suggested answers with discussion related to how those answers resulted. The most important thing to realize is whether your answer works. You should figure out the implications of the answers here and what the effects are from any different answers you may come up with.

Lab 13.1 What Triggers Are

Answer the following questions.

Database Trigger

In this exercise, you need to determine the trigger firing event, its type, and so on, based on the `CREATE` clause of the trigger. Consider the following `CREATE` clause:

```
CREATE TRIGGER student_au
AFTER UPDATE ON STUDENT
FOR EACH ROW
WHEN (NVL(NEW.ZIP, ' ') <> OLD.ZIP)
   Trigger Body…
```

Answer the following questions:

a) Assume a trigger named `STUDENT_AU` already exists in the database. If you use the `CREATE TRIGGER` clause above to modify the existing trigger, what error message is generated? Explain your answer.

   Answer: An error message stating STUDENT_AU name is already used by another object is displayed on the screen. The `CREATE TRIGGER` clause has the ability to create new objects in the database, but it is unable to handle modifications. In order to modify the existing trigger, the reserved word `REPLACE` must be added to the `CREATE TRIGGER` clause. In this case, the old version of the trigger is dropped without warning, and the new version of the trigger is created.

b) If an update statement is issued on the `STUDENT` table, how many times does this trigger fire?

   Answer: The trigger fires as many times as there are rows affected by the triggering event because statement `FOR EACH ROW` is present in the `CREATE TRIGGER` clause. When `FOR EACH ROW` statement is not present in the `CREATE TRIGGER` clause, the trigger fires once for the triggering event. In this case, if the following `UPDATE` statement

```
UPDATE student
   SET zip = '01247'
 WHERE zip = '02189';
```

is issued against the STUDENT table, it updates as many records as there are students with ZIP code 02189. Accordingly, the trigger will fire as many times as there are records affected by this UPDATE statement.

c) How many times does this trigger fire if an UPDATE statement is issued against the STUDENT table, but the ZIP column is not changed?

Answer: The trigger does not fire at all because the condition of the WHEN statement evaluates to FALSE.

The condition

```
(NVL(NEW.ZIP, ' ') <> OLD.ZIP)
```

of the WHEN statement compares the new value of ZIP code to the old value of ZIP code. If the value of the ZIP code is not changed, this condition evaluates to FALSE. As a result, this trigger does not fire if an UPDATE statement does not modify the value of ZIP code for a specified record.

d) Why do you think there is a NVL function present in the WHEN statement of the CREATE TRIGGER clause?

Answer: If an UPDATE statement does not modify the column ZIP, the value of the field NEW.ZIP is undefined. In other words, it is NULL. A NULL value of ZIP cannot be compared with a non-NULL value of ZIP. Therefore, the NVL function is present in the WHEN condition. Note that because the column ZIP has a NOT NULL constraint defined, there is no need to use the NVL function for the OLD.ZIP field. For an UPDATE statement issued against the STUDENT table, there is always a value of ZIP that is currently present in the table.

BEFORE Triggers

In this exercise, you create a trigger on the INSTRUCTOR table that fires before an INSERT statement is issued against the table. The trigger determines the values for the columns CREATED_BY, MODIFIED_BY, CREATED_DATE, and MODIFIED_DATE. In addition, it determines if the value of zip provided by an INSERT statement is valid. Create the following trigger:

For Example *ch13_8a.sql*

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
FOR EACH ROW
DECLARE
   v_work_zip CHAR(1);
BEGIN
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;

   SELECT 'Y'
     INTO v_work_zip
     FROM zipcode
    WHERE zip = :NEW.ZIP;
EXCEPTION
```

```
   WHEN NO_DATA_FOUND
   THEN
      RAISE_APPLICATION_ERROR (-20001, 'Zip code is not valid!');
END;
```

Answer the following questions:

a) If an `INSERT` statement issued against the `INSTRUCTOR` table is missing a value for the column ZIP, does the trigger raise an exception? Explain your answer.

   Answer: Yes, the trigger raises an exception. When an `INSERT` statement does not provide a value for the column ZIP, the value of the data element `:NEW.ZIP` is `NULL`. This value is used in the `WHERE` clause of the `SELECT INTO` statement. As a result, the `SELECT INTO` statement is unable to return data. Therefore, the exception `NO_DATA_FOUND` is raised by the trigger.

b) Modify this trigger so that a more appropriate error message is displayed when an `INSERT` statement is missing a value for the column ZIP.

   Answer: The script should look similar to the following script. All changes are shown in bold.

   For Example *ch13_8b.sql*

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
FOR EACH ROW
DECLARE
   v_work_zip CHAR(1);
BEGIN
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;

   IF :NEW.ZIP IS NULL
   THEN
      RAISE_APPLICATION_ERROR (-20002, 'Zip code is missing!');
   ELSE
      SELECT 'Y'
        INTO v_work_zip
        FROM zipcode
       WHERE zip = :NEW.ZIP;
   END IF;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      RAISE_APPLICATION_ERROR (-20001, 'Zip code is not valid!');
END;
```

   Notice that an `IF-ELSE` statement is added to the body of the trigger. This `IF-ELSE` statement evaluates incoming value of ZIP (`:NEW.ZIP`). If the incoming value of ZIP is `NULL`, the `IF-ELSE` statement evaluates to `TRUE`, and another error message is displayed stating that the value of ZIP is missing. If the IF-ELSE statement evaluates to FALSE, the control is passed to the ELSE part of the statement, and the SELECT INTO statement is executed.

c) Modify this trigger so there is no need to supply the value for the instructor's ID at the time of the INSERT statement.

Answer: The version of the trigger should look similar to the one shown. All changes are highlighted in bold.

For Example *ch13_8c.sql*

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
FOR EACH ROW
DECLARE
   v_work_zip CHAR(1);
BEGIN
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;

   SELECT 'Y'
     INTO v_work_zip
     FROM zipcode
    WHERE zip = :NEW.ZIP;

   :NEW.INSTRUCTOR_ID := INSTRUCTOR_ID_SEQ.NEXTVAL;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
       RAISE_APPLICATION_ERROR (-20001, 'Zip code is not valid!');
END;
```

The original version of this trigger does not derive a value for the instructor's ID. Therefore, an INSERT statement issued against the INSTRUCTOR table has to populate the INSTRUCTOR_ID column as well. The new version of the trigger populates the value of the INSTRUCTOR_ID column, so that the INSERT statement does not have to do it. Generally, it is a good idea to populate columns holding IDs in the trigger because when a user issues an INSERT statement, he or she might not know that an ID must be populated at the time of the insert operation. Furthermore, a user may not know—and more than likely does not know—how to operate sequences to populate the ID.

As mentioned previously, the ability to access sequence via PL/SQL expression is a relatively new feature introduced in Oracle 11*g*. Prior to Oracle 11*g*, you would need to employ the SELECT INTO statement in the body of the trigger in order to populate INSTRUCTOR_ID column. This is illustrated by the code fragment below:

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
…

   SELECT INSTRUCTOR_ID_SEQ.NEXTVAL
     INTO v_instructor_id
```

```
          FROM dual;

        :NEW.INSTRUCTOR_ID := v_instructor_id;
      …
      END;
```

AFTER Triggers

In this exercise, you create a trigger on the `COURSE` table that fires after an `UPDATE` statement is issued against the table. Create the following log table and the trigger on the `COURSE` table:

For Example *ch13_9a.sql*

```
CREATE TABLE course_cost_log
   (course_no NUMBER
   ,cost      NUMBER
   ,modified_by VARCHAR2(30)
   ,modified_date DATE)
/

CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
WHEN (NEW.COST <> OLD.COST)
BEGIN
   INSERT INTO course_cost_log
      (course_no, cost, modified_by, modified_date)
   VALUES
      (:old.course_no, :old.cost, USER, SYSDATE);
END;
/
```

Answer the following questions:

a) Describe the trigger created above.

   Answer: The trigger created above fires after `UPDATE` statement is issued on the `COURSE` table. The `WHEN` clause compares new and old values of the `COST` column, and if these are not the same, the old value of the `COST` column is recorded in the `COURSE_COST_LOG` table along with the course number, user's name, and date of change.

For the next set of questions, execute example ch13_9a.sql, and add a new course to the COURSE table as follows:

```
INSERT INTO course
   (course_no, description, created_by, created_date, modified_by, modified_date)
VALUES
   (999, 'Test Course', user, sysdate, user, sysdate);
COMMIT;
```

Note that the `INSERT` statement above does not provide value for `COST` column. To correct it, issue the following `UPDATE` statement:

```
UPDATE course
   SET cost = 0
 WHERE course_no = 999;
```

```
COMMIT;
```

b) Check how many records are in the COURSE_COST_LOG table? Explain your findings.

Answer: There are no records in the COURSE_COST_LOG table. Even though the UPDATE statement modifies COST column, the condition in the WHEN clause evaluates to FALSE. This is because the old value of the COST column is NULL and it cannot be compared to the new non-value of the COST column. As a result, even though the value of the COST column has changed, there is no record written in the COURSE_COST_LOG table.

c) How would you change the trigger so that it records all changes to the COST column? In other words, if COST column is updated from NULL to non-NULL value or vice versa, the appropriate record is created in the COURSE_COST_LOG table.

Answer: The new version of the trigger should look similar to the following. Modified statements are highlighted in bold.

For Example *ch13_9b.sql*

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
WHEN (NVL(NEW.COST, -1) <> NVL(OLD.COST, -1))
BEGIN
   INSERT INTO course_cost_log
      (course_no, cost, modified_by, modified_date)
   VALUES
      (:old.course_no, :old.cost, USER, SYSDATE);
END;
```

Note that this version of the trigger employs NVL function in the WHEN condition. This guarantees that if the course cost is changed from NULL to some value or vice versa, it will be properly recorded in the log table. In addition, the NULL value of cost is defaulted to –1 because it is an unlikely value for a course cost.

Autonomous Transaction

Answer the following questions:

a) What is an autonomous transaction?

Answer: Autonomous transaction is an independent transaction started by another transaction that is usually referred to as main transaction. In other words, autonomous transaction may issue various DML statements and commit or roll them back, without committing or rolling back the DML statements issued by the main transaction.

b) How would you define an autonomous transaction?

Answer: Autonomous transaction is defined with the AUTONOMOUS_TRANSACTION pragma that is placed in the declaration portion of a trigger.

c) What are some of the reasons to employ an autonomous transaction?

Answer: As stated previously, autonomous transaction is fully independent of its main transaction. As a result, it enables you to log information even if the main transaction is rolled back. For example, you may decide to log data changes in the ENROLLMENT table even when they are rolled back. In such case, you may define a row-level AFTER trigger on the ENROLLMENT table that employs autonomous transaction, and thus is able to record these data changes.

Lab 13.2 Types of Triggers

Answer the following questions.

Row and Statement Triggers

In this exercise, you create a trigger that fires before an INSERT statement is issued against the COURSE table. Create the following trigger:

For Example *ch13_10a.sql*

```
CREATE OR REPLACE TRIGGER course_bi
BEFORE INSERT ON COURSE
FOR EACH ROW
BEGIN
   :NEW.COURSE_NO     := COURSE_NO_SEQ.NEXTVAL;
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
END;
```

Answer the following questions:

a) What type of trigger is created on the COURSE table (row or statement)? Explain your answer.

Answer: The trigger created on the COURSE table is a row trigger because the CREATE TRIGGER clause contains the statement FOR EACH ROW. It means this trigger fires every time a record is added to the COURSE table.

b) Based on the answer you provided for question (a), explain why this particular type is chosen for the trigger.

Answer: This trigger is a row trigger because its operations depend on the data in the individual records. For example, for every record inserted into the COURSE table, the trigger calculates the value for the column COURSE_NO. All values in this column must be unique, because it is defined as a primary key. A row trigger guarantees every record added to the COURSE table has a unique number assigned to the COURSE_NO column.

c) When an INSERT statement is issued against the COURSE table, which actions are performed by the trigger?

Answer: First, the trigger assigns a number derived from the sequence COURSE_ NO_SEQ to the variable v_course_no. Second, the values containing the current user's name and date are assigned to the fields CREATED_BY, MODIFIED_BY, CREATED_DATE, and MODIFIED_DATE of the :NEW pseudorecord.

d) Modify this trigger so that if there is a prerequisite course supplied at the time of the insert, its value is checked against the existing courses in the COURSE table.

Answer: The trigger you created should look similar to the following trigger. Newly added statements are highlighted in bold.

For Example *ch13_10b.sql*

```
CREATE OR REPLACE TRIGGER course_bi
BEFORE INSERT ON COURSE
FOR EACH ROW
DECLARE
   v_prerequisite COURSE.COURSE_NO%TYPE;
BEGIN
   IF :NEW.PREREQUISITE IS NOT NULL
   THEN
      SELECT course_no
        INTO v_prerequisite
        FROM course
       WHERE course_no = :NEW.PREREQUISITE;
   END IF;

   :NEW.COURSE_NO     := COURSE_NO_SEQ.NEXTVAL;
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
       RAISE_APPLICATION_ERROR (-20002, 'Prerequisite is not valid!');
END;
```

Notice that because the PREREQUISITE is not a required column, or, in other words, there is no NOT NULL constraint defined against it, the IF statement validates the existence of the incoming value. Next, the SELECT INTO statement validates that the prerequisite already exists in the COURSE table. If there is no record corresponding to the prerequisite course, the NO_DATA_FOUND exception is raised and the error message “Prerequisite is not valid!” is raised. Once this version of the trigger is created, the INSERT statement

```
INSERT INTO COURSE (description, cost, prerequisite)
VALUES ('Test Course', 0, 9999);
```

causes the following error:

```
SQL Error: ORA-20002: Prerequisite is not valid!
ORA-06512: at "STUDENT.COURSE_BI", line 20
ORA-04088: error during execution of trigger 'STUDENT.COURSE_BI'
```

INSTEAD OF Triggers

In this exercise, you create a view `STUDENT_ADDRESS` and an `INSTEAD OF` trigger that fires when an INSERT statement is issued against the view. Create the following view along with the `INSTEAD OF` trigger:

For Example *ch13_11a.sql*

```
CREATE VIEW student_address
    AS
       SELECT s.student_id, s.first_name, s.last_name, s.street_address, z.city, z.state
             ,z.zip
         FROM student s
         JOIN zipcode z
           ON (s.zip = z.zip);
/

CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
BEGIN
   INSERT INTO STUDENT
      (student_id, first_name, last_name, street_address, zip, registration_date
      ,created_by, created_date, modified_by, modified_date)
   VALUES
      (:NEW.student_id, :NEW.first_name, :NEW.last_name, :NEW.street_address, :NEW.zip
      ,SYSDATE, USER, SYSDATE, USER, SYSDATE);
END;
/
```

Issue the following `INSERT` statements:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street', 'New York'
       ,'NY', '10019');

INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street', 'New York'
       ,'NY', '12345');
```

Answer the following questions:

a) What output is produced after each `INSERT` statement is issued?

Answer: The first `INSERT` statement completes successfully. Whereas, the second `INSERT` statement causes the following error:

```
ORA-02291: integrity constraint (STUDENT.STU_ZIP_FK) violated - parent key not found
ORA-06512: at "STUDENT.STUDENT_ADDRESS_INS", line 2
ORA-04088: error during execution of trigger 'STUDENT.STUDENT_ADDRESS_INS'
```

b) Explain why the second `INSERT` statement causes an error.

Answer: The second `INSERT` statement causes an error because it violates the foreign key constraint on the `STUDENT` table. The value of the ZIP code provided in the `INSERT` statement does not have a corresponding record in the `ZIPCODE` table. Since `ZIP` column of the

STUDENT table has a foreign key constraint STU_ZIP_FK defined on it, each time a record is inserted into the STUDENT table, the incoming value of zip code is checked by the system in the ZIPCODE table. If there is a corresponding record, the INSERT statement against the STUDENT table does not cause errors. For example, the first INSERT statement is successful because the ZIPCODE table contains a record corresponding to the value of ZIP '10019'. The second INSERT statement causes an error because there is no record in the ZIPCODE table corresponding to the value of ZIP '12345'.

c) Modify the trigger so that it checks the value of the ZIP code provided by the INSERT statement against the ZIPCODE table and raises an error if there is no such value.

Answer: The trigger should look similar to the following. Newly added statements are highlighted in bold.

For Example *ch13_11b.sql*

```
CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
DECLARE
   v_zip VARCHAR2(5);
BEGIN
   SELECT zip
     INTO v_zip
     FROM zipcode
    WHERE zip = :NEW.ZIP;

   INSERT INTO STUDENT
      (student_id, first_name, last_name, street_address, zip, registration_date
      ,created_by, created_date, modified_by, modified_date)
   VALUES
      (:NEW.student_id, :NEW.first_name, :NEW.last_name, :NEW.street_address
      ,:NEW.zip, SYSDATE, USER, SYSDATE, USER, SYSDATE);

EXCEPTION
   WHEN NO_DATA_FOUND
   THEN
      RAISE_APPLICATION_ERROR (-20002, 'Zip code is not valid!');
END;
```

In this version of the trigger, the incoming value of ZIP code is checked against the ZIPCODE table via the SELECT INTO statement. If the SELECT INTO statement does not return any rows, the NO_DATA_FOUND exception is raised and the error message stating 'ZIP code is not valid!' is raised. Once this version of the trigger is created, the second INSERT statement produces output as follows:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street', 'New York'
       ,'NY', '12345');

ORA-20002: Zip code is not valid!
ORA-06512: at "STUDENT.STUDENT_ADDRESS_INS", line 19
ORA-04088: error during execution of trigger 'STUDENT.STUDENT_ADDRESS_INS'
```

d) Modify the trigger so that it checks the value of the ZIP code provided by the INSERT statement against the ZIPCODE table. If there is no corresponding record in the ZIPCODE table, the trigger should create a new record for the given value of zip before adding a new record to the STUDENT table.

Answer: This version of the trigger should look similar to the following. All changes are shown in bold.

For Example *ch13_11c.sql*

```
CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
DECLARE
   v_zip VARCHAR2(5);
BEGIN
   BEGIN
      SELECT zip
        INTO v_zip
        FROM zipcode
       WHERE zip = :NEW.zip;
   EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
         INSERT INTO ZIPCODE
            (zip, city, state, created_by, created_date, modified_by, modified_date)
         VALUES
            (:NEW.zip, :NEW.city, :NEW.state, USER, SYSDATE, USER, SYSDATE);
   END;
   INSERT INTO STUDENT
      (student_id, first_name, last_name, street_address, zip, registration_date
      ,created_by, created_date, modified_by, modified_date)
   VALUES
      (:NEW.student_id, :NEW.first_name, :NEW.last_name, :NEW.street_address
      ,:NEW.zip, SYSDATE, USER, SYSDATE, USER, SYSDATE);
END;
```

Just like in the previous version, the existence of the incoming value of ZIP code is checked against the ZIPCODE table via the SELECT INTO statement. When a new value of ZIP code is provided by the INSERT statement, the SELECT INTO statement does not return any rows and causes the NO_DATA_FOUND exception. As a result, the INSERT statement against the ZIPCODE table is executed. Next, control is passed to the INSERT statement against the STUDENT table.
It is important to realize that the SELECT INTO statement and the exception-handling section have been placed in the inner block. This placement ensures that once the exception NO_DATA_FOUND is raised the trigger does not terminate but proceeds with its normal execution.
Once this trigger is created, the second INSERT statement completes successfully:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street', 'New York'
       ,'NY', '12345');
```

```
1 row created.
```

Try It Yourself

The projects in this section are meant to have you use all of the skills that you have acquired throughout this chapter. Here are some exercises that will help you test the depth of your understanding.

1. Create or modify a trigger on the `ENROLLMENT` table that fires before an `INSERT` statement. Make sure all columns that have `NOT NULL` and foreign key constraints defined on them are populated with their proper values.

   Answer: The trigger should look similar to the following:

   For Example *ch13_12a.sql*

```
CREATE OR REPLACE TRIGGER enrollment_bi
BEFORE INSERT ON ENROLLMENT
FOR EACH ROW
DECLARE
   v_valid NUMBER := 0;
BEGIN
   SELECT COUNT(*)
     INTO v_valid
     FROM student
    WHERE student_id = :NEW.STUDENT_ID;

   IF v_valid = 0
   THEN
      RAISE_APPLICATION_ERROR (-20000, 'This is not a valid student');
   END IF;

   SELECT COUNT(*)
     INTO v_valid
     FROM section
    WHERE section_id = :NEW.SECTION_ID;

   IF v_valid = 0
   THEN
      RAISE_APPLICATION_ERROR (-20001, 'This is not a valid section');
   END IF;

   :NEW.ENROLL_DATE   := SYSDATE;
   :NEW.CREATED_BY    := USER;
   :NEW.CREATED_DATE  := SYSDATE;
   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
END;
```

   Consider this trigger. It fires before the `INSERT` statement on the `ENROLLMENT` table. First, it validates the values provided for student ID and section ID. If one of the IDs is invalid, the exception is raised and the trigger is terminated. As a result, the `INSERT` statement would causes an error. If both student and section IDs are found in the STUDENT and SECTION tables, respectively, the ENROLL_DATE, CREATED_DATE, and MODIFIED_DATE are populated with current date, and columns CREATED_BY and MODIFIED_BY are populated with current user name. Consider the following INSERT statement:

```
INSERT INTO enrollment (student_id, section_id)
VALUES (777, 123);
```

The value 777 in this INSERT statement does not exist in the STUDENT table and therefore is invalid. As a result, this statement causes the following error:

```
ORA-20000: This is not a valid student
ORA-06512: at "STUDENT.ENROLLMENT_BI", line 11
ORA-04088: error during execution of trigger 'STUDENT.ENROLLMENT_BI'
```

2. Create or modify a trigger on the SECTION table that fires before an UPDATE statement. Make sure that the trigger validates incoming values so that there are no constraint violation errors.

Answer: The trigger should look similar to the following:

For Example *ch13_13a.sql*

```
CREATE OR REPLACE TRIGGER section_bu
BEFORE UPDATE ON SECTION
FOR EACH ROW
DECLARE
   v_valid NUMBER := 0;
BEGIN
   IF :NEW.INSTRUCTOR_ID IS NOT NULL
   THEN
      SELECT COUNT(*)
        INTO v_valid
        FROM instructor
       WHERE instructor_id = :NEW.instructor_ID;

      IF v_valid = 0
      THEN
         RAISE_APPLICATION_ERROR (-20000, 'This is not a valid instructor');
      END IF;
   END IF;

   :NEW.MODIFIED_BY   := USER;
   :NEW.MODIFIED_DATE := SYSDATE;
END;
```

This trigger fires before the UPDATE statement on the SECTION table. First, it checks if there is a new value for an instructor ID with the help of an IF-THEN statement. If the IF-THEN statement evaluates to TRUE, the instructor's ID is checked against the INSTRUCTOR table. If a new instructor ID does not exist in the INSTRUCTOR table, the exception is raised, and the trigger is terminated. Otherwise, all columns with NOT NULL constraints are populated with their respected values.

Note that this trigger does not populate CREATED_BY and CREATED_DATE columns with the new values. This is because when record is updated, the values for these columns do not change as they reflect when this record was added to the SECTION table.
Consider the following UPDATE statement:

```
UPDATE section
   SET instructor_id = 220
 WHERE section_id = 79;
```

The value 220 in this UPDATE statement does not exist in the INSTRUCTOR table and therefore is invalid. As a result, this UPDATE statement when run causes an error:

```
ORA-20000: This is not a valid instructor
ORA-06512: at "STUDENT.SECTION_BU", line 13
ORA-04088: error during execution of trigger 'STUDENT.SECTION_BU'
```

Next, consider another UPDATE statement that does not cause any errors:

```
UPDATE section
     SET instructor_id = 105
   WHERE section_id = 79;

1 row updated.
```