Triggers

Audit trails and further constraints

What is a trigger?

- An action that is taken when some event occurs.
- The events we will address are Data Manipulation events:
  - INSERT, UPDATE and DELETE operations.

Piggyback

- The operation works as normal on the table in the database, but it now has a trigger, piggybacking on it.

- The **TRIGGER** can
  - log events or constrain them.
- While the **INSERT, UPDATE or DELETE**
  - manipulate the table as intended.

Before or after

- For the moment, we will concentrate on AFTER triggers. These act after the operation has completed.
- After triggers can see all old and new data, but cannot change the data being used by the operation.
  - The 'after' lends the letter 'A' to the trigger name.

Reasons for triggers

- Triggers can be used for:
  - Checking constraints
    - I don’t want a student to register for modules if they are already registered for 60 ECTS credits worth.
  - Logging user actions
    - Who updated a staff member’s salary? When?
  - Other reasons
    - Automatically generate derived column values
    - Put constraints on a transaction.

Firstly, Logging actions.

- The triggers we’ll look at today apply to:
    - INSERT
    - UPDATE and
    - DELETE
- To log these commands we need a log table.

To log actions.

- When logging user actions, in general, we need to know:
  - What table is being changed.
  - What the change is (INSERT, UPDATE or DELETE).
  - Who has made the change.
  - When was the change made.
- *This information tells us a little about the transaction.*
  - *We will need to add more later.*

Logging actions

- To log these actions, we must
  1. Set up a table into which the **TRIGGER** can insert rows.
  2. Attach a **TRIGGER** to an **OPERATION** on a TABLE.
     - i.e. we attach a trigger to
       - An INSERT on a STOCK table.
       - An UPDATE on a STOCK table.
       - A DELETE on a STOCK table

Part 1 – Set up the log table.

- The log table fields will be:
  - Table name (of the table being manipulated)
  - Operation used
  - User name
  - System date.

```
CREATE TABLE LOGTABLE
  (
    tabnam VARCHAR2(20),
    opname CHAR(3) CHECK (opname IN ('INS','UPD','DEL')),
    Uname  VARCHAR2(20),
    Sdate DATE
  );
```

The trigger text

```
CREATE OR REPLACE TRIGGER stock_AI
AFTER INSERT ON stock
BEGIN
   INSERT INTO logtable VALUES(
   'STOCK', 'INS',TO_CHAR(USER),SYSDATE);
END;
```

- Run the above to get ‘trigger created’.
  - The compile follows the same format as that for functions and procedures. The compiled trigger will show up in the connections menu under ‘triggers’/

To execute the trigger

```
INSERT INTO STOCK (STOCK_CODE, STOCK_DESCRIPTION,
  STOCK_LEVEL, UNIT_PRICE) VALUES
  ('A222','Binder',40,3.00);
1 row created.
SQL>Select * from logtable;
   TABNAM  OPN UNAME   SDATE
   __________________________
   STOCK   INS POBYRNE 01-APR-16
```

Piggyback

- The operation works as normal on the table in the database, but it now has a trigger, piggybacking on it.
    - The **TRIGGER** adds a row to the **logtable**.
    - While the **INSERT** adds a row to the **stock** table,
- Other operations
    - Triggers can be adapted to work on UPDATEs o DELETEs.

Table and Row Triggers

- The triggers covered so far are **statement** level triggers.
- They ‘fire’ every time a statement is executed.
  - Delete from stock where supplierid = 501;
  - This deletes multiple rows, but only logs 1 DELETE.
- To itemise each row manipulation, you must use ROW triggers.
  - To do this, add the line `'FOR EACH ROW'` to the code.

More specific audit trails

- When an INSERT … FOR EACH ROW… trigger is active, it has access to the new values that the INSERT is trying to put into the table.
- These values can be accessed by preceding the attribute name with **`:new.`**
  - E.g. to get the value that the insert is trying to put into the stock_code column in the table, we can reference **`:new.stock_code`**

Data available to INSERT trigger

Data available to the INSERT trigger

The STOCK_INSERT trigger

:new.stock_code
:new.stock_description
:new.unit_price
:new.unitcostprice
:new.stock_level
:new.reorder_level
:new.supplier_id

The `INSERT into STOCK` operation

Using data in the log

- If I want to store the key value, as well as the table name, I can add a column to the log table as follows:
  - `alter table logtable add keyfield char(6);`
- I can now change the trigger, to add the new keyfield value to the log.
- See next slide.

Row level audit trigger

```
CREATE OR REPLACE TRIGGER stock_air
AFTER INSERT ON stock
FOR EACH ROW
BEGIN
  INSERT INTO logtable VALUES(
    'STOCK',
    'INS',
    TO_CHAR(USER),
    SYSDATE,
    :new.stock_code);
END;
```

Update triggers

- Statement level trigger

```
CREATE OR REPLACE TRIGGER stock_au
BEFORE UPDATE ON stock
BEGIN
   INSERT INTO logtable
   (tabnam, opname, usr, sdate)
   VALUES(
   'STOCK','UPD',TO_CHAR(USER),SYSDATE);
END;
```

Data available to UPDATE trigger

Data available to the UPDATE trigger

The STOCK_UPDATE trigger

The `UPDATE STOCK` operatio

:new.stock_code
:new.stock_description
:new.unit_price
:new.unitcostprice
:new.stock_level
:new.reorder_level
:new.supplier_id

:old.stock_code
:old.stock_description
:old.unit_price
:old.unitcostprice
:old.stock_level
:old.reorder_level
:old.supplier_id

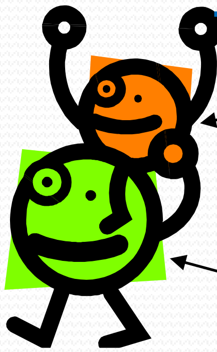
More detail on updates

- To get more detail on updates, it is likely that we will need to have a specific log file for updates.
- For example, if the price changes, we may wish to record:
  - The key value of the row whose price changed.
  - The old price
  - The new price
- for each record.

Data available to DELETE trigger

Data available to the DELETE trigger

:old.stock_code
:old.stock_description
:old.unit_price
:old.unitcostprice
:old.stock_level
:old.reorder_level
:old.supplier_id



The STOCK_AD trigger

DELETE STOCK **operation**

The Delete trigger can only see what values were there previously.

Combining triggers

```
create or replace TRIGGER report_stock
  BEFORE
    INSERT OR
    UPDATE OF stock_level OR
    DELETE
  ON stock
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('stock_level') THEN
      DBMS_OUTPUT.PUT_LINE('Updating stock_level');
      WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END CASE;
  END;
```

Order in which triggers fire

- If two or more triggers are defined for the same statement on the same table, then they fire in this order:
  - All BEFORE STATEMENT triggers
  - All BEFORE EACH ROW triggers
  - All AFTER EACH ROW triggers
  - All AFTER STATEMENT triggers
- *See follows | precedes if you want to have > 1 trigger with the same timing point (i.e. BEFORE / AFTER)*

Triggers Part 2

Constraining and changing values using triggers

Declarative constraints

- Where possible, the state of the data in the database is guarded by declarative constraints. E.G:

```
CREATE TABLE Sorder (
SupplierOrderNo    NUMBER(7)                   NOT NULL,
SupplierOrderDate DATE DEFAULT sysdate NOT NULL,
DeliveredDate      DATE,
Supplier_Id        NUMBER(7)                   NOT NULL,
CONSTRAINT  XPKSO PRIMARY KEY (SupplierOrderNo),
CONSTRAINT DATECHECK
  CHECK(DELIVEREDDATE >SUPPLIERORDERDATE),
CONSTRAINT  FROMSUPP FOREIGN KEY (Supplier_Id)
REFERENCES Supplier(Supplier_Id));
```

Limitations on CHECK Constraint

- The CHECK constraint:
  - It must be a Boolean expression evaluated using only the values in the row being inserted or updated, and
  - It cannot contain sub-queries, sequences or certain Oracle functions.
- We require to be able to:
  - Check data that is not in the current row, either in this table or possibly other tables.
  - Cause the operation to fail if it breaks the rules.

Triggering failure

- In PL/SQL there are a few issues to consider:
  1. The database is designed to protect the data, not the user.
  2. Errors are handled through exceptions.
  3. Exceptions can be propagated back to the user through unhandled exceptions.
     1. We do this by
        1. Declaring our own exceptions
        2. and using 'raise_application_error'

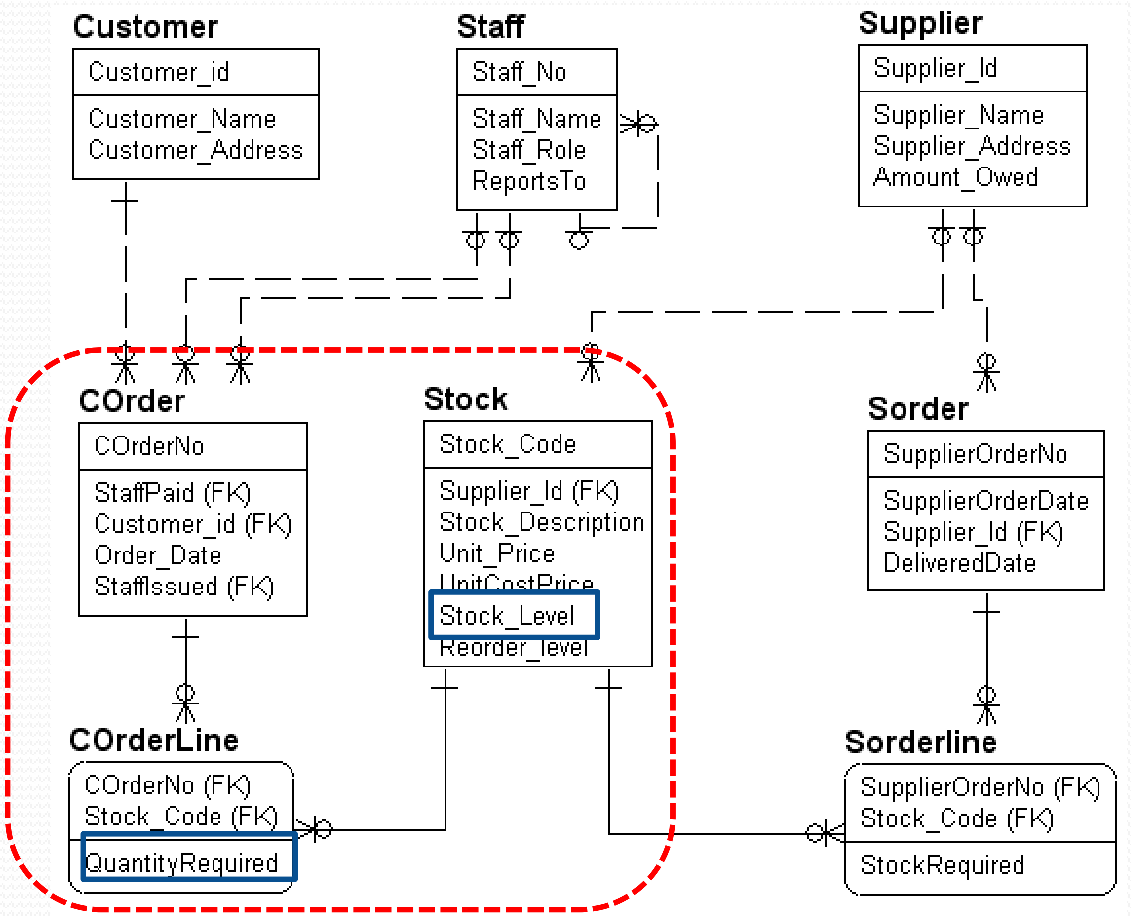
To make an operation fail…

- To cause an operation to fail,
- the trigger must raise an exception.
- The exception in turn, raises an application error.

Triggers and exceptions

- Using triggers to check to see if an event is within the domain constraints of the system, e.g.:
  - I can’t sell stock I don’t have.
  - I don’t want a student to register for modules if they are already registered for 60 ECTS credits worth.
- If a trigger finds an error, it needs to inform:
  - The user
  - The DBMS

Recalling the builder schema…



- When a customer order line is added, the customer specifies a quantityrequired.
- Is there enough stock?
  - Check the stock_level in the stock table.

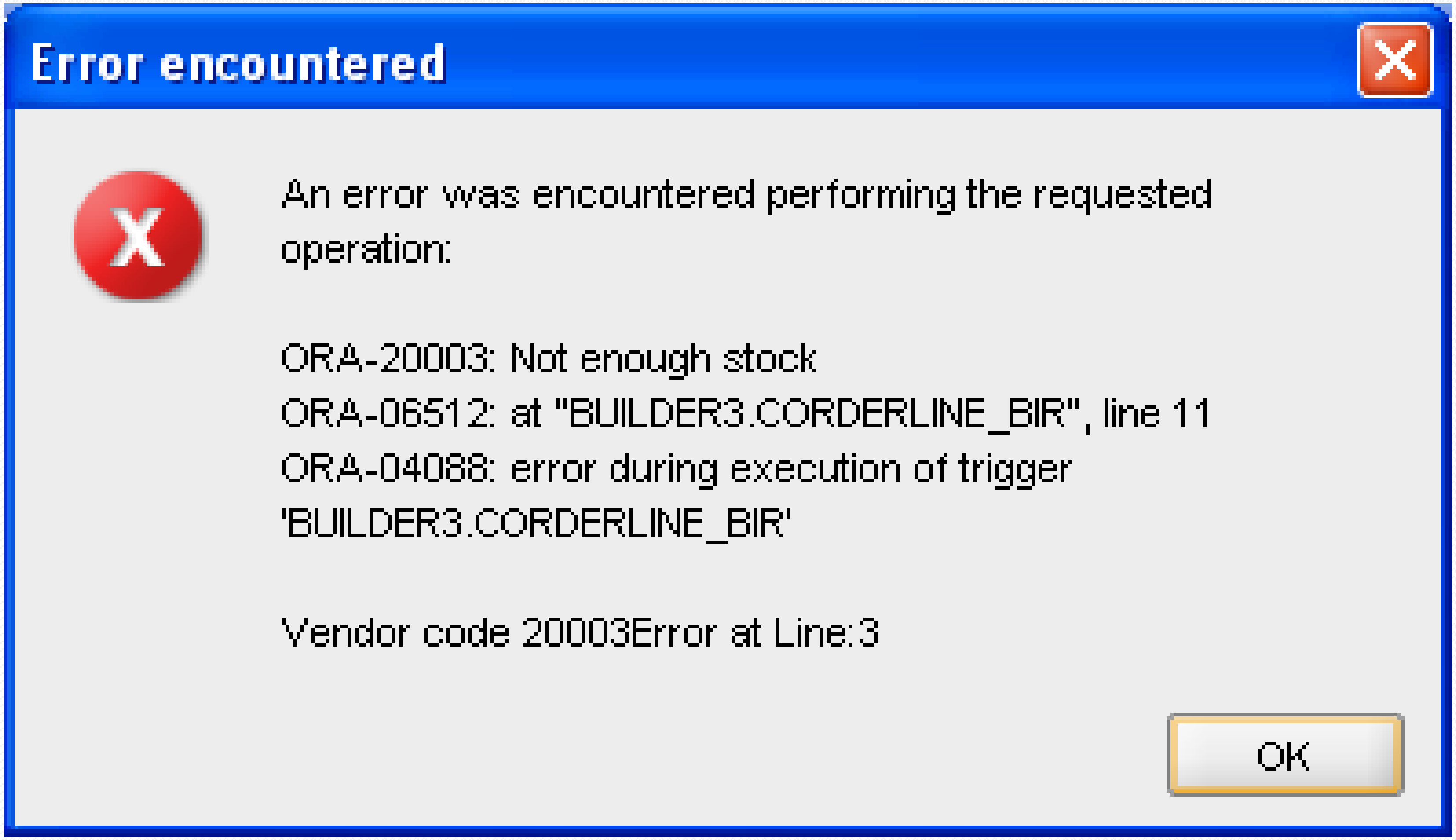
Example trigger

```
CREATE OR REPLACE TRIGGER CORDERLINE_BIR
BEFORE INSERT ON CORDERLINE
FOR EACH ROW
DECLARE
  NOT_ENOUGH_STOCK EXCEPTION;
  QTY INTEGER;
BEGIN
SELECT STOCK_LEVEL INTO QTY FROM STOCK WHERE
  STOCK_CODE = :NEW.STOCK_CODE;
IF QTY - :NEW.QUANTITYREQUIRED < 0 THEN
 RAISE NOT_ENOUGH_STOCK;
END IF;
EXCEPTION
WHEN NOT_ENOUGH_STOCK THEN
  RAISE_APPLICATION_ERROR(-200003,'Not enough stock');
END;
```

Java has an SQLException class that can catch these errors. See http://download.oracle.com/javase/1.4.2/docs/api/java/sql/SQLException.html

How do I test it?

```
select stock_code, stock_level  from stock;
-- told me that B111 has a stock level of 10
select corderno from corder where corderno
  not in (select corderno from corderline);
-- told me that corderno 202 has no order lines.
insert into corderline values
  (202,'B111',11);
```



An error was encountered performing the requested operation:

```
ORA-20003: Not enough stock
ORA-06512: at "BUILDER3.CORDERLINE_BIR", line 11
ORA-04088: error during execution of trigger
'BUILDER3.CORDERLINE_BIR'

Vendor code 20003Error at Line:3
```

Before or after?

- Any trigger can reject a statement.
- Only a ‘before’ statement can amend the values that go into the database.
- If you want to use a trigger to change the contents of an insert or update statement, such as SYSDATE or USER from the dual table, you must do it in a ‘before’ trigger.

Drop a Trigger

- The syntax for a dropping a Trigger is:

```
DROP TRIGGER trigger_name;
```

- For example:
  - If you had a trigger called orders_bir, you could drop it with the following command:

```
DROP TRIGGER orders_bir;
```

Sample triggers for BUILDER

- Insert trigger that could go on corderline:
  - Check there is enough stock before selling.
  - Reject invalid transactions
  - Record reorder requirements
- Update trigger on supplier order delivery date
  - Check that date is not already there
  - Check that date is not in the future.