MET CS 669 Database Design and Implementation for Business
Lab 4 Explanation: Procedural SQL and Concurrency

Lab 4 Explanations For Oracle

This explanation document illustrates how to correctly execute each SQL construct step-by-step for Lab 4, and explains important theoretical and practical details. Before completing a step, read its explanation here first.

Use this explanations document only if you are using Oracle. If you are using SQL Server or PostgreSQL, explanations for those DBMS' are available in a different document in the assignments section of the course.

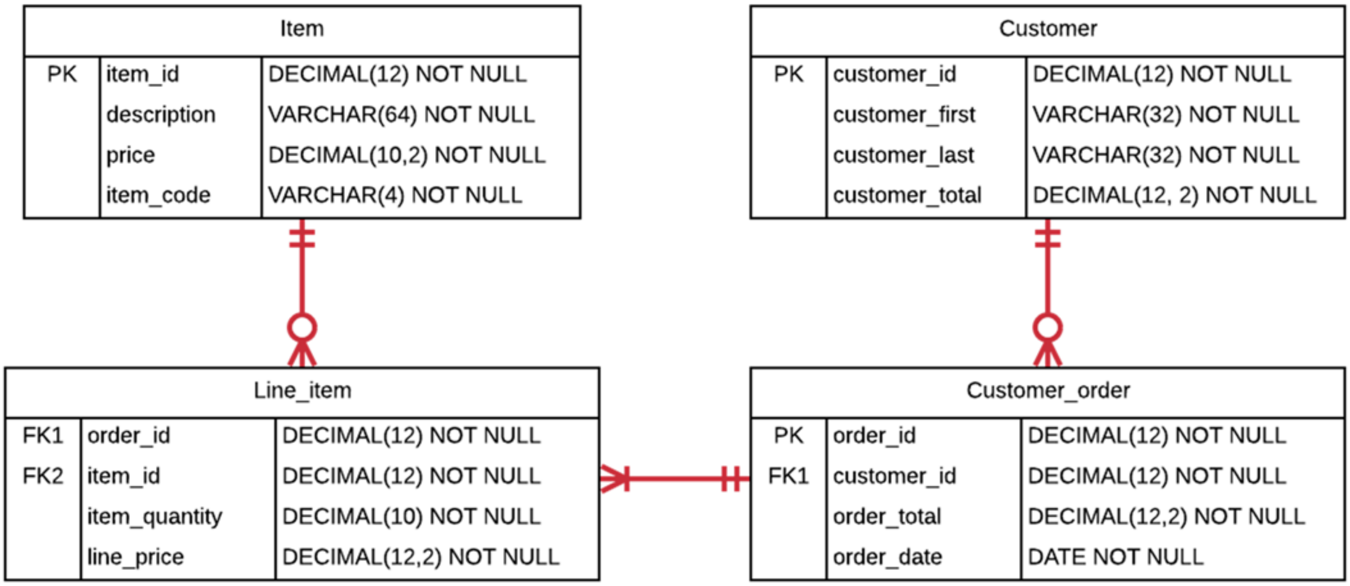
Authored by Warren Mansur.

Table of Contents

Section One – Stored Procedures

Step 1 – Create Table Structure

To help demonstrate how to complete the commands in this section, we work with simplified customer order schema which tracks customers and their orders of items. The schema itself is illustrated below.



Item

PK	item_id	DECIMAL(12) NOT NULL
	description	VARCHAR(64) NOT NULL
	price	DECIMAL(10,2) NOT NULL
	item_code	VARCHAR(4) NOT NULL

Customer

PK	customer_id	DECIMAL(12) NOT NULL
	customer_first	VARCHAR(32) NOT NULL
	customer_last	VARCHAR(32) NOT NULL
	customer_total	DECIMAL(12, 2) NOT NULL

Line_item

FK1	order_id	DECIMAL(12) NOT NULL
FK2	item_id	DECIMAL(12) NOT NULL
	item_quantity	DECIMAL(10) NOT NULL
	line_price	DECIMAL(12,2) NOT NULL

Customer_order

PK	order_id	DECIMAL(12) NOT NULL
FK1	customer_id	DECIMAL(12) NOT NULL
	order_total	DECIMAL(12,2) NOT NULL
	order_date	DATE NOT NULL

The Item table contains items that can be purchased, with a primary key, a description of the item, a price, and an item_code which is an identifier by which the item can be referenced. The Customer table contains basic information on customers such as their first and last name, and a total balance which they owe. The Customer_order table contains basic information about an order itself, including a reference to the customer who placed the order, the sum total for the order, and the date the order was placed. The Line_item table contains information on individual lines in the order, including a reference to the item that was purchased, the quantity that was purchased, and the total amount for that line (for example, if an item costs $10 and 2 of them were purchased, the total amount for the line would be $20).

We create the customer odder schema illustrated above in its entirety, including all primary and foreign key constraints, with the SQL below.

Here’s the code we use for creating the schema.

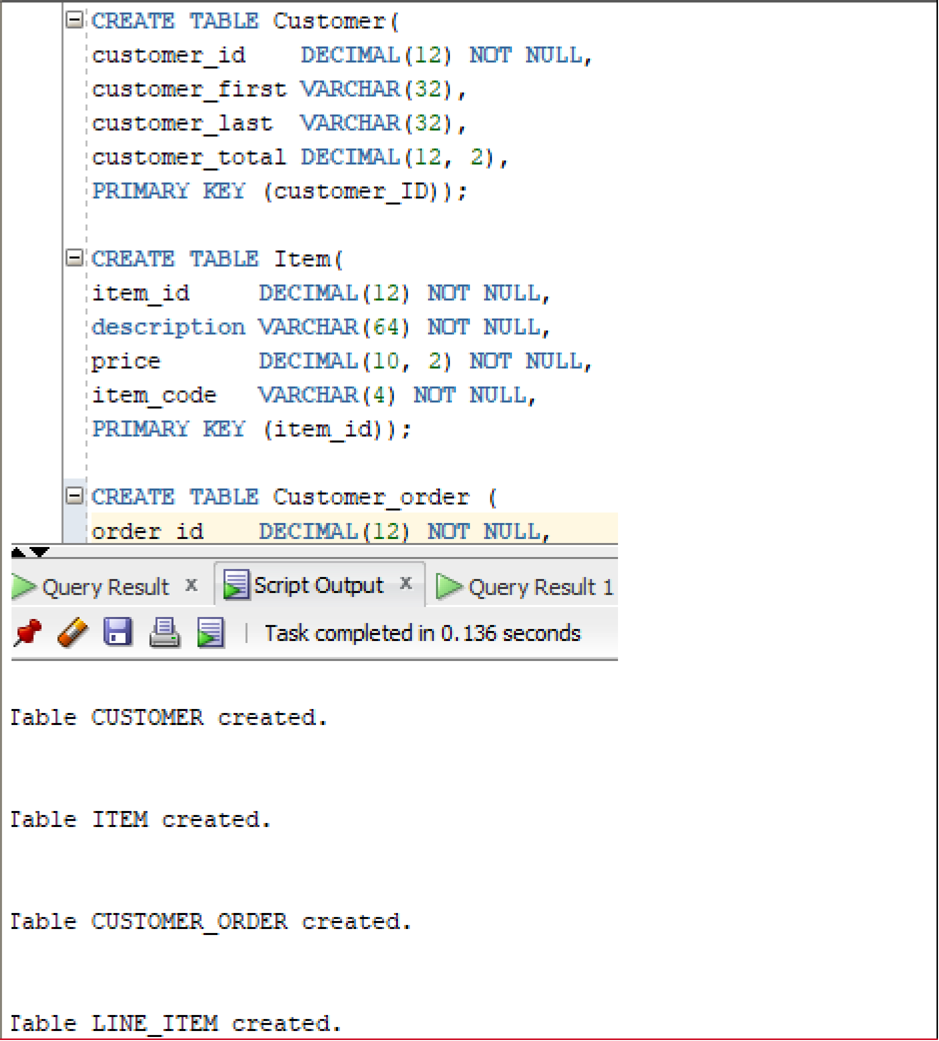
Code: Creating Customer Order Schema

```
CREATE TABLE Customer(
customer_id    DECIMAL(12) NOT NULL,
customer_first VARCHAR(32),
customer_last  VARCHAR(32),
customer_total DECIMAL(12, 2),
PRIMARY KEY (customer_ID));

CREATE TABLE Item(
item_id     DECIMAL(12) NOT NULL,
description  VARCHAR(64) NOT NULL,
price       DECIMAL(10, 2) NOT NULL,
item_code   VARCHAR(4) NOT NULL,
PRIMARY KEY (item_id));

CREATE TABLE Customer_order (
order_id    DECIMAL(12) NOT NULL,
customer_id DECIMAL(12) NOT NULL,
order_total DECIMAL(12,2) NOT NULL,
order_date  DATE NOT NULL,
PRIMARY KEY (ORDER_ID),
FOREIGN KEY (CUSTOMER_ID) REFERENCES customer);

CREATE TABLE Line_item(
order_id      DECIMAL(12) NOT NULL,
item_id       DECIMAL(12) NOT NULL,
item_quantity DECIMAL(10) NOT NULL,
line_price    DECIMAL(12,2) NOT NULL,
PRIMARY KEY (ORDER_ID, ITEM_ID),
FOREIGN KEY (ORDER_ID) REFERENCES customer_order,
FOREIGN KEY (ITEM_ID) REFERENCES item);
```

Below are screenshots of creating the schema.

Screenshot: Creating the Customer Order Schema

Oracle



```
CREATE TABLE Customer(
customer_id    DECIMAL(12) NOT NULL,
customer_first VARCHAR(32),
customer_last  VARCHAR(32),
customer_total DECIMAL(12, 2),
PRIMARY KEY (customer_ID));

CREATE TABLE Item(
item_id     DECIMAL(12) NOT NULL,
description VARCHAR(64) NOT NULL,
price       DECIMAL(10, 2) NOT NULL,
item_code   VARCHAR(4) NOT NULL,
PRIMARY KEY (item_id));

CREATE TABLE Customer_order (
order_id    DECIMAL(12) NOT NULL,
```

Task completed in 0.136 seconds

```
Table CUSTOMER created.


Table ITEM created.


Table CUSTOMER_ORDER created.


Table LINE_ITEM created.
```

Our next step is to create a sequence for each table. A *sequence* is a database object capable of generating unique primary key values, and is the preferred mechanism for doing so. Sequences generate unique whole numbers, starting with the first, and incrementing to the next number each time a new value is needed. The database guarantees a sequence will not generate the same number twice, thus making the values unique and suitable for primary keys.

Why use a sequence when we can hardcode values like 1 and 2 in our insert statements? There are many reasons. First, inserts often happen over many years as applications live on and on, so it is impractical to hardcode every value. Additionally, real-world databases oftentimes have millions of rows in some of the more used tables. Again, it is not practical to hardcode millions of values in such instances. We are looking for automation, so we do not want a human being to be asked for the primary and foreign key values every time an application needs to insert more rows. In short, dynamically generating primary key values is critical to modern database operation, and sequences are the preferred means of doing so.

There are a few advanced options with sequences which are available but not typically used for primary key configurations. Many sequences are configured to start at 1, and increment upward by 1 each time a new value is needed; however, the starting value and incrementing value can be changed if needed (such as starting at 10 and incrementing by 10, for example). Sequences can be set to cycle, which means once they reach their maximum value, they reset back to the minimum value again. This means once a cycle occurs, the sequence will be regenerating numbers already generated, making them non-unique. Typically for primary keys, the sequence is not configured to cycle. Instead, we make sure the primary key is large enough to hold all values, current and future.

The SQL syntax for creating a basic sequence is straightforward:

CREATE SEQUENCE <sequencename> START WITH 1

Because we need one sequence per table, a convention I recommend for sequence names is to name it like this:

tablename_seq

For example, if we had a Person table and that table needed a sequence, we would name the sequence "person_seq".

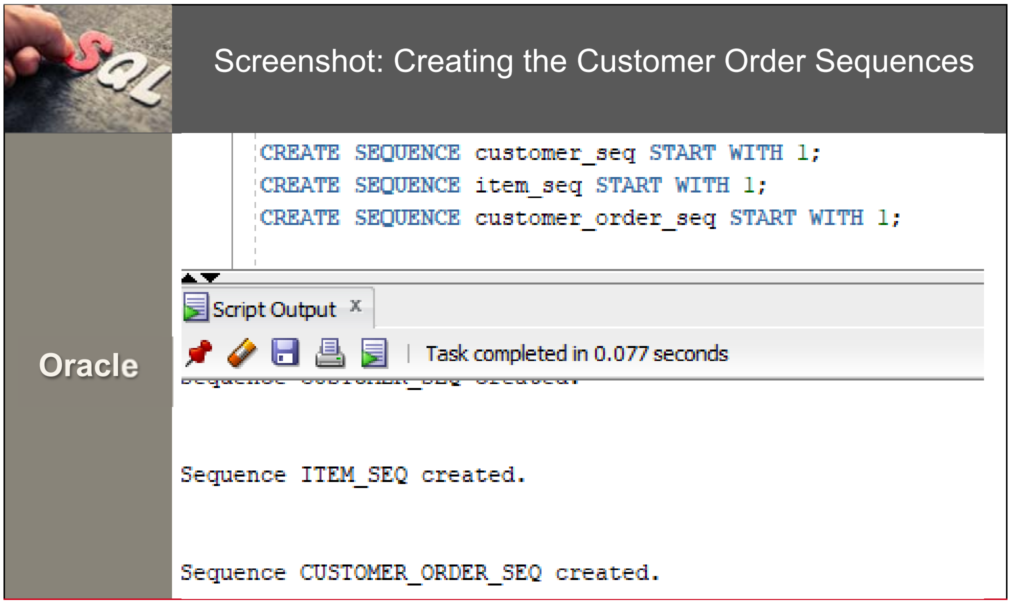
Below are the sequences I create for the customer order schema.

Code: Creating Customer Order Sequences

```
CREATE SEQUENCE customer_seq START WITH 1;
CREATE SEQUENCE item_seq START WITH 1;
CREATE SEQUENCE customer_order_seq START WITH 1;
```

The three sequences are for the Customer, Item, and Customer_order tables, respectively. Line_item does not need a sequence, because it uses a composite primary key consisting of other tables' keys. That is, we do not need to generate unique values for the Line_item table since it does not use a synthetic primary key of its own.

Creating them looks as follows.



Screenshot: Creating the Customer Order Sequences

```
CREATE SEQUENCE customer_seq START WITH 1;
CREATE SEQUENCE item_seq START WITH 1;
CREATE SEQUENCE customer_order_seq START WITH 1;
```

```
Sequence CUSTOMER_SEQ created.


Sequence ITEM_SEQ created.


Sequence CUSTOMER_ORDER_SEQ created.
```

As you can see, creating sequences for tables is straightforward.

Step 2 – Populate Tables

You can interact with a sequence in one of two ways – obtaining the next value of the sequence, and obtaining the current value. Obtaining the next value generates the next unique number. The next value is typically requested when a new primary key value is needed for a table. Sequences are safe to use concurrently. This means even if hundreds or thousands of transactions are running simultaneously that use the same sequence, the sequence generates unique numbers for them all. Obtaining the current value does not generate the next unique number, but instead returns the last number requested. The current value is typically used when a primary key value has already been obtained, and that same value is needed again so it can be inserted into a different table as a foreign key.

When sequences are involved, the order of inserts becomes quite important. The reason is, if we use a sequence to insert a new row, and then a referencing table must have a foreign key to that row. We would want to use the sequence's current value to create the foreign key, so wouldn't want to change the value.

For example, in the customer order schema, Customer_order references Customer. So rather than inserting all customers followed by all customer orders, we would insert one customer, followed by all of that customer's orders. Then we would insert the second customer, followed by the second customer's orders, and so on. This way we can make use of the sequence's *current value* to insert the foreign keys. Let's look at an example of inserting the first customer and order.

Code: Inserting First Customer and Order

```
--Insert first customer and order.
INSERT INTO customer VALUES(customer_seq.nextval,'John','Smith',0);
INSERT INTO customer_order VALUES(customer_order_seq.nextval,customer_seq.currval,
506,CAST('18-DEC-2005' AS DATE));
```

Since this is the first time we are using a sequence in this lab, let's examine this code closely. The first line inserts John Smith as a customer, and uses *customer_seq.nextval* as the means of obtaining the primary key value from *customer_seq*, in lieu of hardcoding a primary key value (such as "1" or "2"). The *nextval* keyword instructs the database to retrieve the next unique value from that sequence. It will retrieve "1" since the sequence was just created.

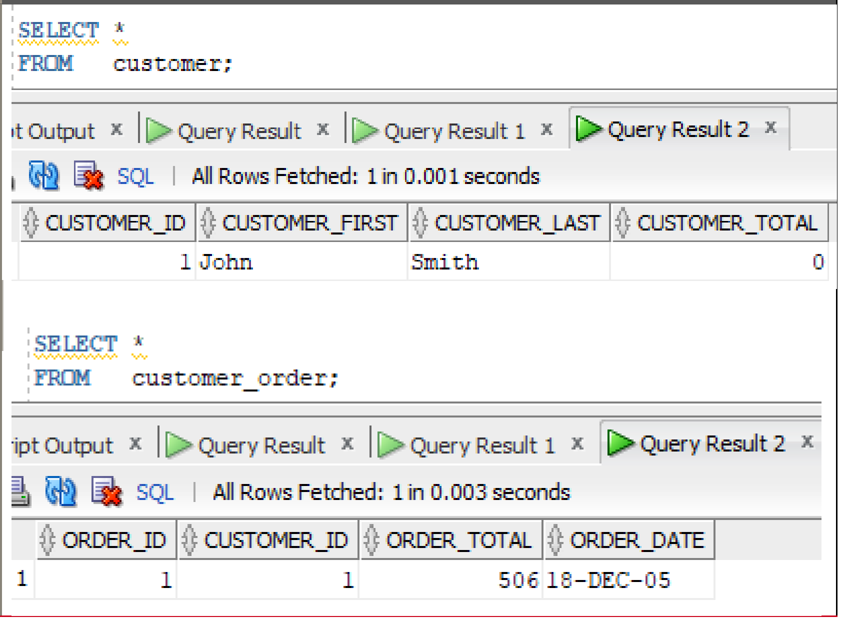
The second insert is an order for John Smith, making use of two sequences. The primary key of the table is set using *customer_order_seq.nextval*, similar to the first line. The foreign key referencing back to Customer is set using *customer_seq.currval*. The *currval* keyword is used to retrieve the latest retrieved value. For example, if the first line's use of *customer_seq.nextval* generated "1" as the value, then calls to *customer_seq.currval* would also return "1".

This is why it is important to only insert one customer at-a-time. We need to generate a new value for *customer_seq.nextval* in order to create the new customer, and then we need to re-use that value for all of that customer's orders using *customer_seq.currval*. If we had inserted all customers first, then we'd need some other means of looking up the foreign key values when inserting the orders.

Let's examine the two tables after the inserts above have been executed.

Screenshots: Tables After Firsts Inserted

Oracle



```
SELECT *
FROM   customer;
```

All Rows Fetched: 1 in 0.001 seconds

CUSTOMER_ID	CUSTOMER_FIRST	CUSTOMER_LAST	CUSTOMER_TOTAL
1	John	Smith	0

```
SELECT *
FROM   customer_order;
```

All Rows Fetched: 1 in 0.003 seconds

	ORDER_ID	CUSTOMER_ID	ORDER_TOTAL	ORDER_DATE
1	1	1	506	18-DEC-05

Because we had just created the sequence and it starts at 1, John Smith's customer_id value was generated as 1 when using *customer_seq.nextval.* Similarly, the first order has a primary key value of 1 for the same reason. And, the order successfully references back to John Smith's primary key value of 1, by using *customer_seq.currval* keyword for the foreign key value. So, we have seen an example where, with careful ordering of our inserts and judicious use of the *nextval* and *currval* keywords, we can use sequences for our inserts, instead of hardcoding values.

Before proceeding further, we need to think about how we are going to insert line items and items. Line items reference back to the orders they belong to, in addition to the item they are representing. While we may be able to use the *currval* method to reference back to the order, we can't do the same for the item foreign key, because many different line items may reference the same item. That is, different customers may purchase the same item, so we can't possibly order everything in a way such that using *nextval* and *currval* is sufficient.

We need another method, which is a *subquery lookup*. In short, in lieu of hardcoding an item's primary key value, we'll look it up with a small subquery instead. Continuing on after inserting our first customer and order, we'll go ahead and insert all items with the following code.

Code: Inserting All Items

```
INSERT INTO item VALUES(item_seq.nextval,'Plate',10, 'P001');
INSERT INTO item VALUES(item_seq.nextval,'Bowl',11, 'B002');
INSERT INTO item VALUES(item_seq.nextval,'Knife',5, 'K003');
INSERT INTO item VALUES(item_seq.nextval,'Fork',5, 'F004');
INSERT INTO item VALUES(item_seq.nextval,'Spoon',5, 'S005');
INSERT INTO item VALUES(item_seq.nextval,'Cup',12, 'C006');
```

We insert all items, generating unique primary keys with *item_seq.nextval*. With those inserted, we can now create the line items for the first customer's first order, shown below.

Code: Inserting Line Items for First Order

```
--Create the line items for the first order.
INSERT INTO line_item
VALUES(customer_order_seq.currval, (SELECT item_id FROM item WHERE description='Plate'),10,100);
INSERT INTO line_item
VALUES(customer_order_seq.currval, (SELECT item_id FROM item WHERE description='Spoon'),2,10);
INSERT INTO line_item
VALUES(customer_order_seq.currval, (SELECT item_id FROM item WHERE description='Bowl'),36,396);
```

Let's review these inserts. For the foreign key to customer_order, we use *customer_order_seq.currval* to refer to the last inserted customer order, which is set to the first order (since we inserted only one order thus far). We are already familiar with using *currval* from the prior examples, and this part of the insert is just another example of doing so.

For retrieving the primary key of each item, however, we use the subquery **`(SELECT item_id FROM item WHERE description=<item_description>)`**. Although subqueries in general are the topic of a future lab, for this lab it is enough to know that instead of hardcoding a single value, we can substitute a query that retrieves exactly one value in its place. In this case, each subquery retrieves one item id corresponding to the item that matches the description in the subquery's WHERE clause. For example, the first insert retrieves the item_id for "Plate", meaning that the first line will reference "Plate" in the Item table. The second line item is for spoons, and the third line item is for bowls.

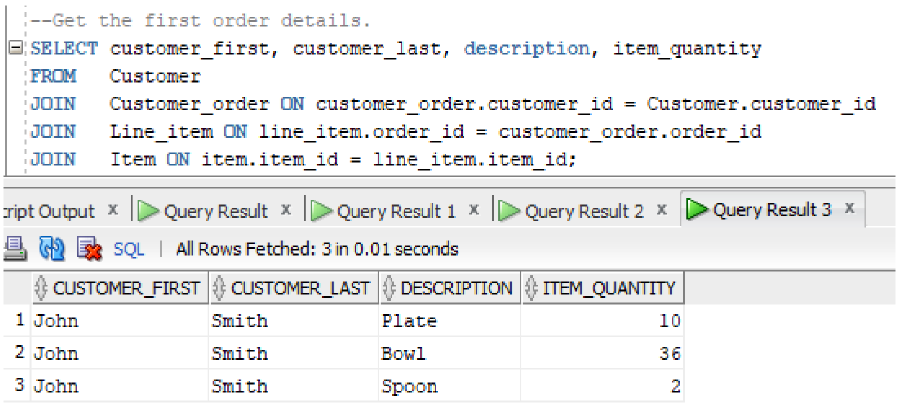
With the first order completely inserted, we can now use the query shown below to view the items in the order.

Code: Viewing First Order

```
--Get the first order details.
SELECT customer_first, customer_last, description, item_quantity
FROM   Customer
JOIN   Customer_order ON customer_order.customer_id = Customer.customer_id
JOIN   Line_item ON line_item.order_id = customer_order.order_id
JOIN   Item ON item.item_id = line_item.item_id;
```

Screenshot: First Order Line Items

Oracle



```
--Get the first order details.
SELECT customer_first, customer_last, description, item_quantity
FROM   Customer
JOIN   Customer_order ON customer_order.customer_id = Customer.customer_id
JOIN   Line_item ON line_item.order_id = customer_order.order_id
JOIN   Item ON item.item_id = line_item.item_id;
```

All Rows Fetched: 3 in 0.01 seconds

	CUSTOMER_FIRST	CUSTOMER_LAST	DESCRIPTION	ITEM_QUANTITY
1	John	Smith	Plate	10
2	John	Smith	Bowl	36
3	John	Smith	Spoon	2

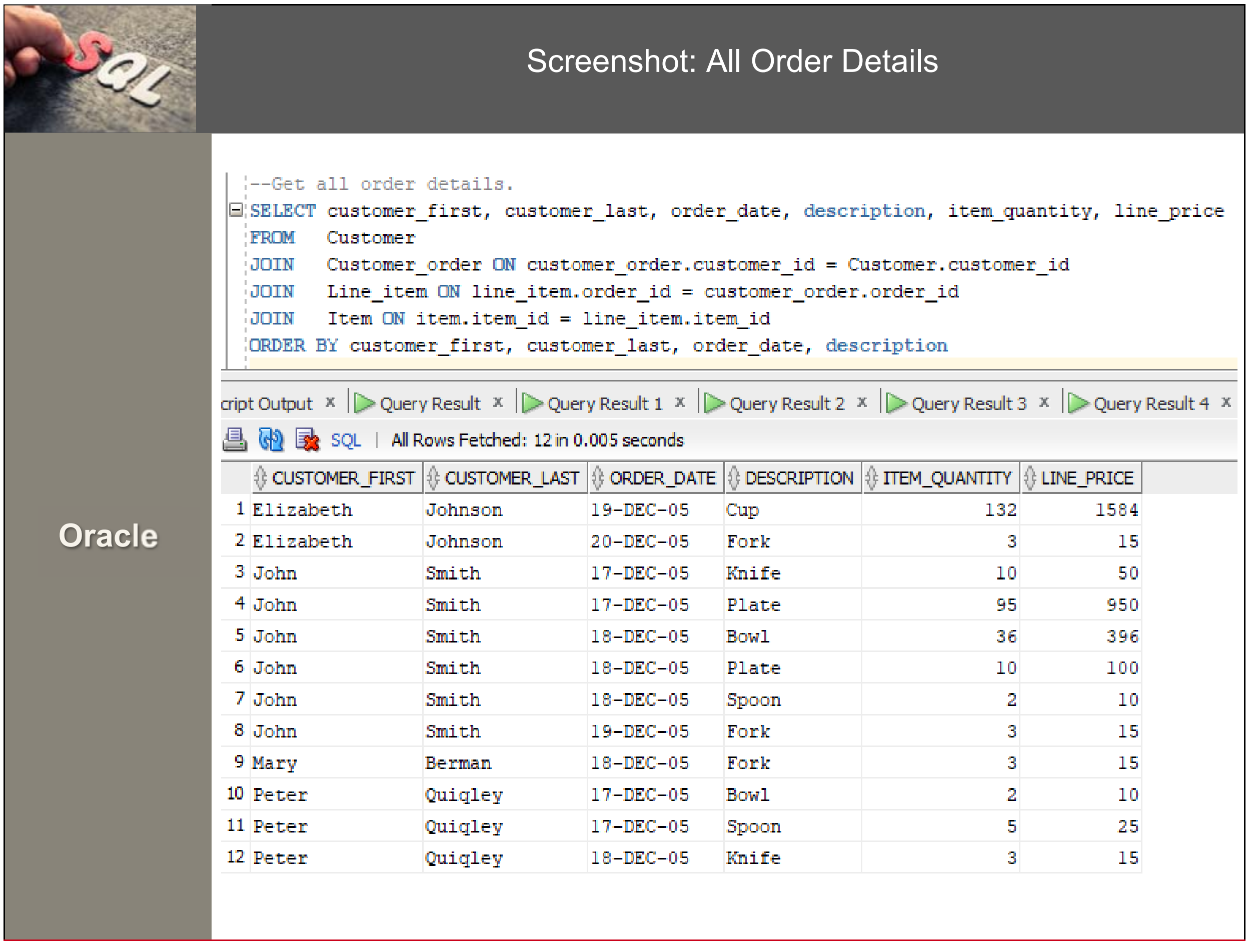
The query results show us that the first order is for a plate, bowl, and spoon, with varying quantities of each.

Below are the remainder of the inserts. These use both methods.

Code: Inserting Remainder of Customer Data

```
--Insert the rest of the customers, orders, and line items.
--John's remaining orders.
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,1000,CAST('17-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Plate'),95,950);
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Knife'),10,50);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,10,CAST('19-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Mary's orders.
INSERT INTO customer VALUES(customer_seq.nextval,'Mary','Berman',0);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,1584,CAST('18-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Elizabeth's orders.
INSERT INTO customer VALUES(customer_seq.nextval,'Elizabeth','Johnson',0);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,15,CAST('19-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Cup'),132,1584);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,15,CAST('20-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Fork'),3,15);
--Peter's orders.
INSERT INTO customer VALUES(customer_seq.nextval,'Peter','Quiqley',0);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,100,CAST('17-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Spoon'),5,25);
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Bowl'),2,10);
INSERT INTO customer_order
VALUES(customer_order_seq.nextval,customer_seq.currval,40,CAST('18-DEC-2005' AS DATE));
INSERT INTO line_item
VALUES(customer_order_seq.currval,(SELECT item_id FROM item WHERE description='Knife'),3,15);
```

The below screenshot captures the important order information, showing all orders.



Screenshot: All Order Details

```
--Get all order details.
SELECT customer_first, customer_last, order_date, description, item_quantity, line_price
FROM   Customer
JOIN   Customer_order ON customer_order.customer_id = Customer.customer_id
JOIN   Line_item ON line_item.order_id = customer_order.order_id
JOIN   Item ON item.item_id = line_item.item_id
ORDER BY customer_first, customer_last, order_date, description
```

All Rows Fetched: 12 in 0.005 seconds

	CUSTOMER_FIRST	CUSTOMER_LAST	ORDER_DATE	DESCRIPTION	ITEM_QUANTITY	LINE_PRICE
1	Elizabeth	Johnson	19-DEC-05	Cup	132	1584
2	Elizabeth	Johnson	20-DEC-05	Fork	3	15
3	John	Smith	17-DEC-05	Knife	10	50
4	John	Smith	17-DEC-05	Plate	95	950
5	John	Smith	18-DEC-05	Bowl	36	396
6	John	Smith	18-DEC-05	Plate	10	100
7	John	Smith	18-DEC-05	Spoon	2	10
8	John	Smith	19-DEC-05	Fork	3	15
9	Mary	Berman	18-DEC-05	Fork	3	15
10	Peter	Quigley	17-DEC-05	Bowl	2	10
11	Peter	Quigley	17-DEC-05	Spoon	5	25
12	Peter	Quigley	18-DEC-05	Knife	3	15

In summary, you have seen two methods that use sequences to retrieve the correct foreign key values. The first method uses *sequence.currval*. This method works when we can control the order of inserts, and we insert the referencing values immediately after the referenced values (such as inserting a customer's orders immediately after inserting the customer). The second method is using a subquery lookup. With the second method, there is less concern about the order of inserts and referencing values don't need to be inserted immediately after referenced values. Instead, the foreign key value can be retrieved with a subquery.

Armed with both of these methods, you can now complete this step.

Step 3 – Create Hardcoded Procedure

To demonstrate something similar, we'll create a stored procedure named "add_customer_harry" that has no parameters and adds a customer named Harry Joker to the customer order schema. Below is code for this procedure.

Code: Creating add_customer_harry Procedure

```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY
IS
BEGIN
  INSERT INTO CUSTOMER
   (customer_id,customer_first,customer_last,customer_total)
   VALUES(customer_seq.nextval, 'Harry', 'Joker', 0);
END;
```

Let us go through code line by line and discuss the meaning.

Line 1: CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY	
	The **CREATE OR REPLACE PROCEDURE** phrase indicates to Oracle that a stored procedure is to be created, and that the creation replaces any existing definition of a stored procedure with the same name. If we instead use the phrase **CREATE PROCEDURE,** then Oracle would create the procedure only if one with the same name does not exist. All of the words in this phrase are SQL *keywords*, meaning that they are words predefined in the SQL language to have a specific meaning. We must use keywords exactly as they are defined by the language in order to tell the language what we want to do. The **ADD_CUSTOMER_HARRY** word is the name of the stored procedure. This name is an *identifier*, which means that the language allows us to define our own name. Oracle does restrict the length of identifiers to be no longer than 30 characters, and has some character restrictions, for example, that identifiers should not contain the "%" character. Within these restrictions we can specify any name we like. Of course, it is best to name a stored procedure reasonably based upon the function it performs. For this procedure, I chose the name **ADD_CUSTOMER_HARRY** because the logic of the procedure inserts a customer named "Harry" into the Customer table. Oracle relaxes many of its restrictions on an identifier if the identifier is quoted; however, it is best not to use an identifier that must be quoted each time it is used, so we stay within the nonquoted identifier guidelines.
Line 2: IS	
	IS is a SQL keyword that is required by the language to define a stored procedure, but otherwise has no significant meaning. SQL is defined to be natural for English speakers, so the full phrase **CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY IS** leads an English speaker to naturally think that the definition of the stored procedure follows the **IS** keyword.
Line 3: BEGIN	

In Oracle, the procedural language always exists within a *block* within SQL. The **BEGIN** keyword is part of the opening definition of a block, and is always accompanied with an **END** keyword which closes the block. We can also embed some nonprocedural (declarative) SQL commands within a block, so that procedural constructs coexist with nonprocedural SQL commands. However, it is important to note that embedding SQL inside of the procedural language is different than using SQL outside of a procedural block. Inside of a procedural block, only certain SQL commands can be embedded, and the commands are expected to fit within the overall flow of the procedural language. Outside of a procedural block, SQL commands are simply executed to produce results, and the commands are not intertwined with procedural constructs. In this case, I use the **BEGIN** keyword to as part of the beginning of the block for the **ADD_CUSTOMER_HARRY** stored procedure.

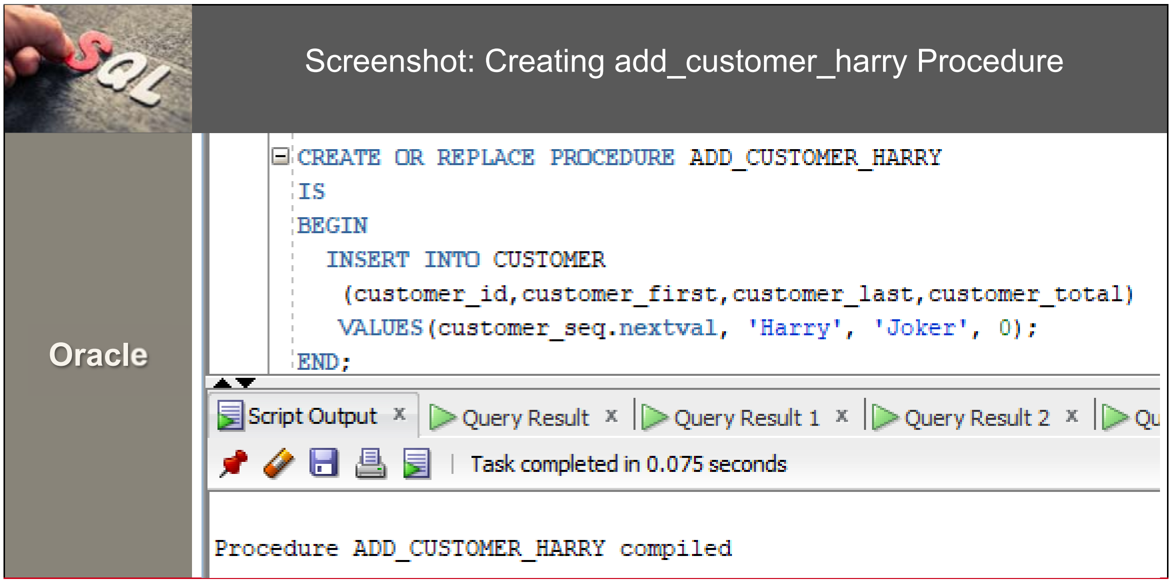
```
Lines 4-6:
INSERT INTO CUSTOMER
  (customer_id,customer_first,customer_last,customer_total)
   VALUES(customer_seq.nextval, ’Harry’, ’Joker’, 0);
```

You might think this command looks familiar, because it does! It is just a standard SQL statement. Wait a minute. The procedural language and the SQL language are two different languages, right? So why is this SQL statement inside of the procedural language block? Simple! Because certain SQL commands can be embedded inside of the procedural language in the right context. In this case, I have embedded an insert statement that inserts a new customer “Harry” into the Customer table. This way, when you execute this stored procedure, the stored procedure will insert the new customer on your behalf, without the need for you to type the SQL command yourself.

```
Line 7: END;
```

The **END** keyword tells Oracle that the procedural block, and therefore the stored procedure definition, ends.

Now to be clear, the above code, when executed, creates the stored procedure so that it’s available for use. Below is a screenshot of creating this stored procedure.



Screenshot: Creating add_customer_harry Procedure

```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER_HARRY
IS
BEGIN
  INSERT INTO CUSTOMER
   (customer_id,customer_first,customer_last,customer_total)
   VALUES(customer_seq.nextval, 'Harry', 'Joker', 0);
END;
```

```
Procedure ADD_CUSTOMER_HARRY compiled
```

Notice that the output indicates that the procedure was compiled. This is a technical way for the DBMS to state that the procedure has been processed and is available for use.

Now that we have created the stored procedure, we need to execute it for its code to take effect. We can do so by defining an *anonymous block* in which to execute procedural code, and then invoking the stored procedure within that block. First, let's look at the code to do so.

Code: Executing add_customer_harry Procedure

```
BEGIN
  ADD_CUSTOMER_HARRY;
END;
/
```

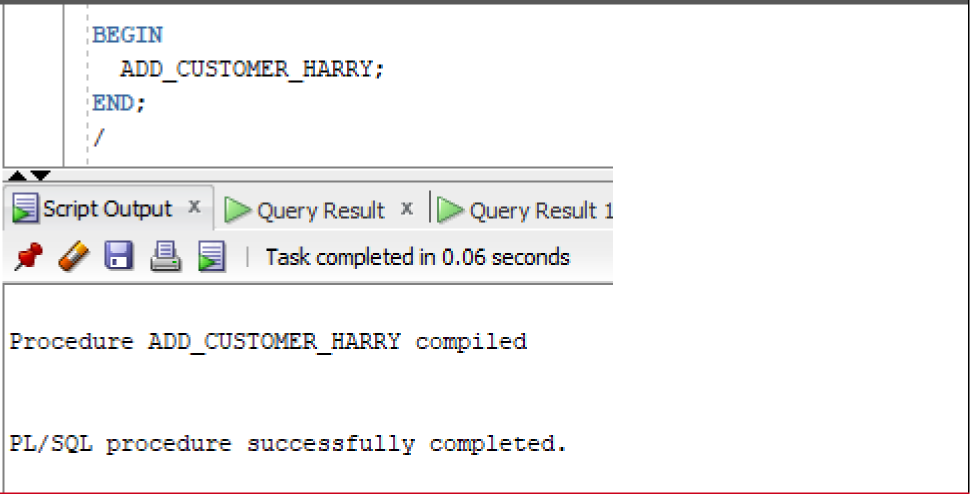
Again, let us visit the meaning of the code line by line.

`Line 1: BEGIN`	
	Just as with a stored procedure, we use the **`BEGIN`** keyword as part of starting the block of procedural code. The difference is that we have not associated this block with any persistent stored module, and so we are creating an *anonymous* procedural block. Anonymous blocks are not durably saved to the database, and can be embedded into scripts to perform logic that does not need to be durably saved, commonly because it is a one-time action.
`Line 2: ADD_CUSTOMER_HARRY;`	
	Here we have simply given the name of the stored procedure followed by a semicolon. In the context of a procedural block, this instructs Oracle to execute the code defined in the stored procedure.
`Line 3: END;`	
	The **`END`** keyword tells Oracle that the anonymous procedural block is ended. At this point, the anonymous procedural block has been defined, but has not yet been executed. This is similar to the way that a stored procedure is first defined and later executed. Why is it necessary to place a semicolon after the **`END`** keyword? By the nature of the language, procedural blocks are defined within the context of the SQL engine, but are actually executed by a separate procedural engine. So from the context of SQL, a block is just another command, and the block needs the semicolon as a statement separator, the same as any other SQL command.
`Line 4: /`	
	The slash (/) tells Oracle to execute the PL/SQL block most recently defined, which in our case is the block defined by the previous three lines of code. The first three lines defined the block, and this line instructs Oracle to execute the block.

Below is a screenshot of executing this code.

Screenshot: Executing add_customer_harry Procedure

Oracle



```
BEGIN
  ADD_CUSTOMER_HARRY;
END;
/
```

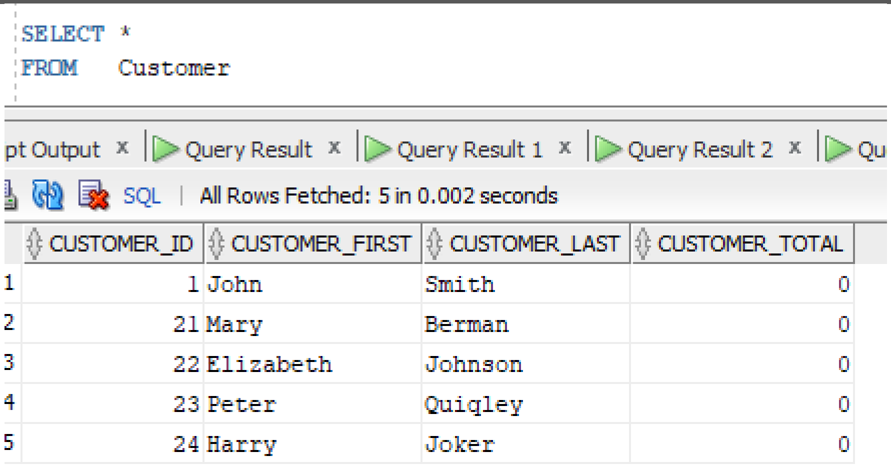
Task completed in 0.06 seconds

```
Procedure ADD_CUSTOMER_HARRY compiled


PL/SQL procedure successfully completed.
```

The output states “PL/SQL procedure successfully completed” to indicate that the code within the anonymous block has executed. We can now select all rows from the Customer table to make sure that our stored procedure inserted a row as we would expect.

Screenshot: Customer Table After Execution

Oracle



```
SELECT *
FROM   Customer
```

All Rows Fetched: 5 in 0.002 seconds

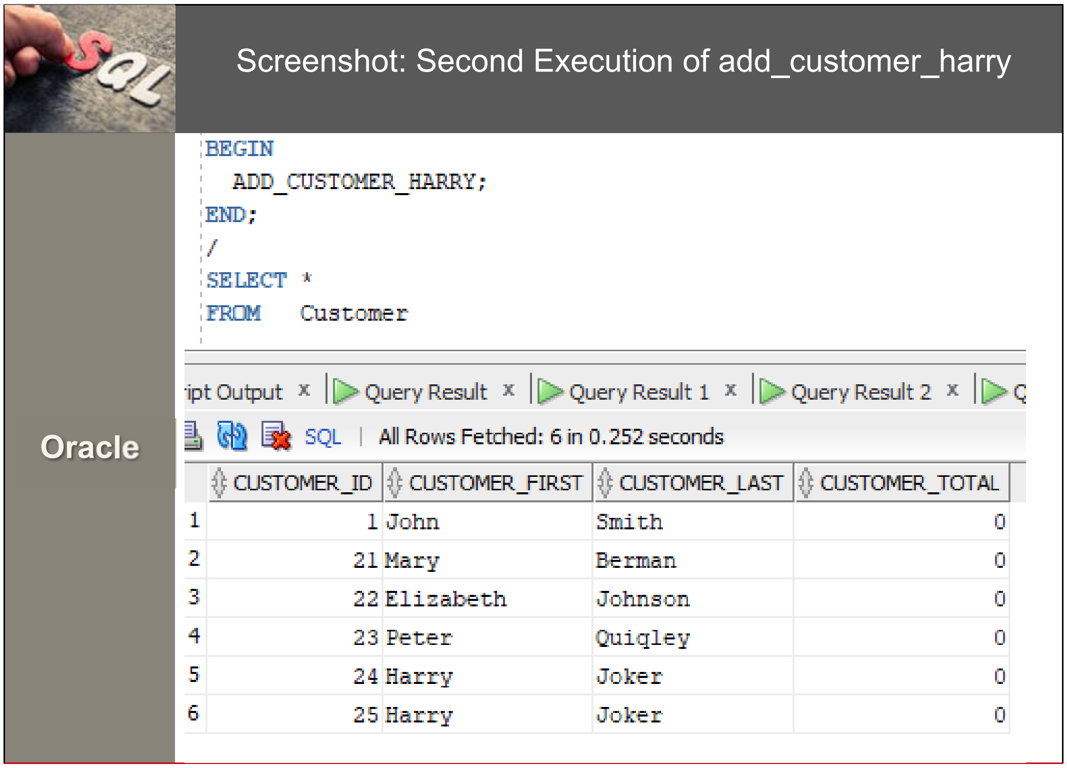
	CUSTOMER_ID	CUSTOMER_FIRST	CUSTOMER_LAST	CUSTOMER_TOTAL
1	1	John	Smith	0
2	21	Mary	Berman	0
3	22	Elizabeth	Johnson	0
4	23	Peter	Quigley	0
5	24	Harry	Joker	0

Sure enough, we see that the customer “Harry Joker” is listed in the table as the last row listed. We have now successfully created and executed a stored procedure! Note that by default stored procedures execute within the current transaction, so we would need to commit the transaction if we want to durably add it to the database.

You can now create similar code to complete this step.

Step 4 – Create Reusable Procedure

Before we create a reusable procedure, let us look at what happens if we attempt to execute the add_customer_harry procedure a second time. Realistically the ADD_CUSTOMER_HARRY stored procedure can be meaningfully executed only once. Inside the procedure, we placed the literal value "Harry" for the customer_first column, the literal value "Joker" for the customer_last column, and the literal value "0" for the customer_total column. This placement is termed "hardcoding" by computer programmers, a term which means that a value is embedded directly into the source of a program, instead of obtaining the value dynamically. Executing it a second time would result in inserting the same person again, just with a different primary key. This is illustrated below.



Screenshot: Second Execution of add_customer_harry

```
BEGIN
  ADD_CUSTOMER_HARRY;
END;
/
SELECT *
FROM   Customer
```

All Rows Fetched: 6 in 0.252 seconds

	CUSTOMER_ID	CUSTOMER_FIRST	CUSTOMER_LAST	CUSTOMER_TOTAL
1	1	John	Smith	0
2	21	Mary	Berman	0
3	22	Elizabeth	Johnson	0
4	23	Peter	Quigley	0
5	24	Harry	Joker	0
6	25	Harry	Joker	0

Notice that Harry Joker has been inserted a second time with the next primary key value. This obviously causes many issues, anomalies resulting from data redundancy perhaps being the most significant. Which record should Harry's orders be tied to? Will there be orders tied to both records? What if Harry has a name change? Should we delete one of these records, and if so, which one? These are valid questions!

The root problem is that the stored procedure is not reusable. Every time it is invoked, it will add Harry Joker and only Harry Joker. The stored procedure is only useful for one execution, which essentially defeats the point of creating the logic in a stored procedure. One significant purpose of a stored procedure is to encapsulate logic so it can be invoked again and again by name. This stored procedure has not achieved that purpose. You would get similar results when you execute your stored procedure a second time.

You might also wonder why there is a gap in values for *customer_id* between John and Mary, because John has a primary key value of 1, and Mary has 21. This is because of the way Oracle optimizes its use of sequences in a concurrent environment. Oracle reserves a certain number of sequence values for a session to optimize performance, so that if multiple requests are made, it does not need to update the database file for every one. Because this explanation lab was written over a period of a few days, the session had expired between "John

Smith" and "Mary Berman". So, Oracle jumped to a higher value, 21, when inserting Mary. Although this is the case, we must remember that *customer_id uses synthetic values that bear no meaning for the data other than unique identification*. It doesn't matter if a customer has a primary key value of 1 or 21 or 1,000. As long as it's unique and permanent, it can be used to identify the same customer again and again.

It is best to make our stored procedures reusable, so that they can be executed wherever the logic contained in them is needed. The fact that our ADD_CUSTOMER_HARRY stored procedure cannot meaningfully be executed multiple times makes it less valuable as a resource. To achieve reusability, instead of hardcoding literal values in a procedure, we use placeholders which instruct the DBMS to use whatever value is given to the stored procedure when it is executed. These placeholders are termed *parameters*. At a minimum, a parameter has a name, which is an identifier by which it is referred, and a datatype, which determines the set of legal values that can be assigned to the parameter.

For example, instead of hardcoding the value "Harry" for the *first_name* column, we can define a *first_name_arg* parameter with a datatype of "VARCHAR". The parameter the particular value can be specified when the stored procedure is executed. Below is an ADD_CUSTOMER stored procedure that makes use of parameters and is therefore reusable, allowing us to add any customer rather than just one specific customer. Comments next to the parameters help explain their purpose.

Code: Creating add_customer Procedure

```
CREATE OR REPLACE PROCEDURE ADD_CUSTOMER( -- Create a new customer
      first_name_arg IN VARCHAR, -- The new customer's first name
      last_name_arg IN VARCHAR)  -- The new customer's last name
IS
BEGIN
   INSERT INTO CUSTOMER (customer_id,customer_first,customer_last,customer_total)
   VALUES(customer_seq.nextval,first_name_arg,last_name_arg,0);
   -- We start the customer with zero balance.
END;
```

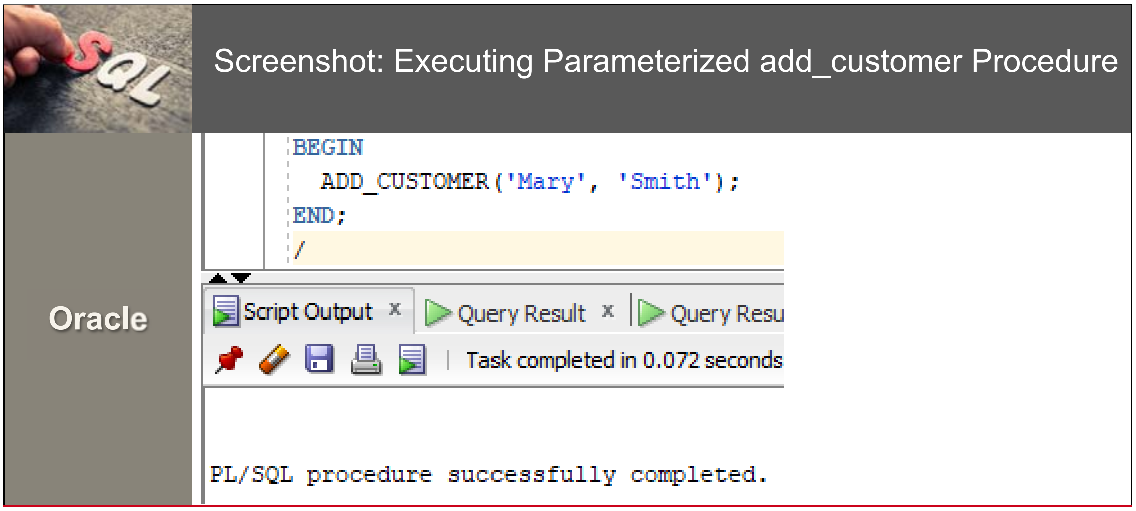
Notice that instead of hardcoding particular values in the insert statement, the parameter names are referenced instead, particularly in the `VALUES(customer_seq.nextval,first_name_arg,last_name_arg,0)` part of the insert statement. The first_name_arg and last_name_arg parameters are used in place of hardcoded "Harry" and "Joker" values. Essentially, this is instructing the SQL engine to insert whatever values are passed into the stored procedure when it is executed.

When the stored procedure is executed, the parameter values are specified by the executor. Example code for executing this stored procedure is below. Notice that the parameters are specified within parentheses, and separated by a comma.

Code: Executing add_customer Procedure

```
BEGIN
  ADD_CUSTOMER('Mary', 'Smith');
END;
/
```

Notice that "('Mary', 'Smith')" part of the stored procedure call which specifies what parameters to use. The order in which the parameters appear matters, as this ordering is correlated with the ordering the parameters are declared in the stored procedure. Since "Mary" comes first, it's matched to first_name_arg, and "Smith" is matched to last_name_arg. Using this approach, you need to know the order in which the parameters are declared in the stored procedure in order to execute the stored procedure. Executing the stored procedure gives us the same confirmation we saw with the prior execution of the “add_customer_harry” procedure, as shown below.



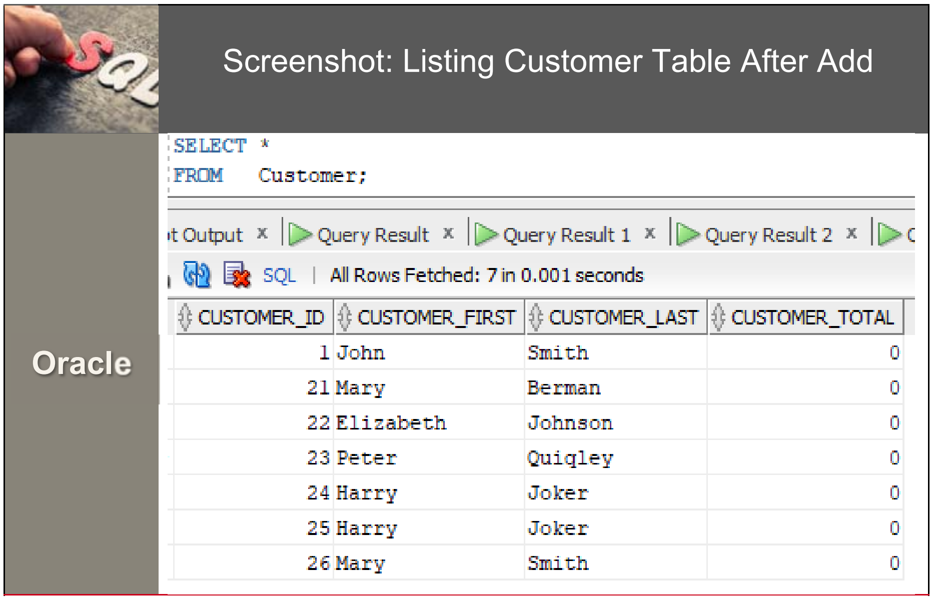
Screenshot: Executing Parameterized add_customer Procedure

```
BEGIN
  ADD_CUSTOMER('Mary', 'Smith');
END;
/
```

Task completed in 0.072 seconds

```
PL/SQL procedure successfully completed.
```

Just to make sure Mary Smith made it in, we’ll list out our Customer table again, illustrated below.



Screenshot: Listing Customer Table After Add

```
SELECT *
FROM   Customer;
```

All Rows Fetched: 7 in 0.001 seconds

CUSTOMER_ID	CUSTOMER_FIRST	CUSTOMER_LAST	CUSTOMER_TOTAL
1	John	Smith	0
21	Mary	Berman	0
22	Elizabeth	Johnson	0
23	Peter	Quigley	0
24	Harry	Joker	0
25	Harry	Joker	0
26	Mary	Smith	0

Notice that Mary Smith is now listed in the table. More importantly, we could add many more customers using this stored procedure just by changing the parameter values given to the stored procedure!

Hopefully this gives you an idea of the usefulness of parameterized stored procedures. You can code the logic once, then execute the stored procedure whenever you need it. For example, the logic we put into this procedure is:

- to use customer_seq to generate a unique primary key value for the new customer.
- to use the parameters given for the first and last name.
- to initialize the new customer’s balance to 0.

We could do the above manually again and again each time we insert a new customer. But doing so is error prone and less convenient. Of course, the logic above is fairly simple so only saves us minimal work; however, you will end up putting much more logic than this into more complex procedures, including parameter validations. You can now use a similar approach to address this step.

Step 5 – Create Deriving Procedure

You learned in the prior step how to create reusable stored procedures by using parameters, so using a variable is the new skill for this step. The basic concepts of variables are not too complex. A variable is a named placeholder that can store a value, and can later be referenced by name to retrieve the stored value.

Let's take an example from the customer order schema we have been using throughout this section. What if, instead of hardcoding the item code for an item, we wanted the database to assign it a unique value? What we could do is, create a variable, calculate the item code and store it in the variable, then reference the variable when inserting into the item table. Let's first illustrate this in pseudo-code so that you understand the concepts.

Pseudocode for Basic Variable Use

```
1: Declare variable v_item_code as a character string
2: Calculate a unique value and store it in the v_item_code variable
3: Insert whatever value is in the v_item_code variable into the item table
```

In line 1 in the pseudocode, the v_item_code variable is declared. A *variable declaration* identifies the existence, name, and datatype of the variable. In programmatic SQL (and also in many programming languages), a variable cannot be used unless it is first declared. Its name identifies how the variable will be later referenced, and its datatype indicates what kind of value it can store (such as character string, number, date, etc …)

In line 2 in the pseudocode, the variable is assigned a value. A *variable assignment* places a value into the variable. Referencing the variable later will use the value assigned.

In line 3, the variable is used by referencing it by name. A *variable reference* uses whatever value is in the variable. Of course, the references to the variable are what makes a variable useful, since simply declaring one and assigning a value to it would not be useful alone.

Now that you understand the pseudocode, let's look at the stored procedure code then analyze the lines.

Code: Creating add_item Procedure

```
CREATE OR REPLACE PROCEDURE ADD_ITEM(
  p_description IN VARCHAR, -- The item's description
  p_price IN DECIMAL)       -- The item's price
IS
  v_item_code VARCHAR(4);   --Declare a variable to hold an item_code value.
BEGIN
   --Calculate the item_code value and put it into the variable.
   v_item_code := SUBSTR(p_description, 1, 1) || ROUND(DBMS_RANDOM.VALUE(0,999), 0);

   --Insert a row with the combined values of the parameters and the variable.
   INSERT INTO ITEM (item_id, description, price, item_code)
   VALUES(item_seq.nextval, p_description, p_price, v_item_code);
END;
```

First, you'll notice that the procedure is named "add_item" since it allows for adding an item to the database. Next, you'll notice that two of the four values needed in the Item table – description, and price – have corresponding parameters. The executor will decide what these values are whenever the stored procedure is invoked (you have already witnessed this in the prior step).

Notice the `v_item_code VARCHAR(4);` code that sits between the `IS` and the `BEGIN` statements. This line is the variable declaration, where we indicate the variable exists (by the existence of the declaration), give the variable its name (v_item_code), and its datatype (VARCHAR(4)). We give it that datatype since that is the same datatype as found in the table. This variable declaration sets up the variable so it can be assigned values and its values can be retrieved.

Next, you'll notice the `v_item_code := SUBSTR(p_description, 1, 1) || ROUND(DBMS_RANDOM.VALUE(0,999), 0)` line. There are several pieces of code here you may not recognize, but don't let that keep you from understanding the basic fact that this is the line that sets the value for the variable. What we are setting it to is the first character of the description, followed by a random 3-digit number. For example, if the item description is "Napkin", then the item code would start with "N" since that is the first letter, followed by a random 3-digit number. If the database randomly selects 867 for example, then the item code would be "N867".

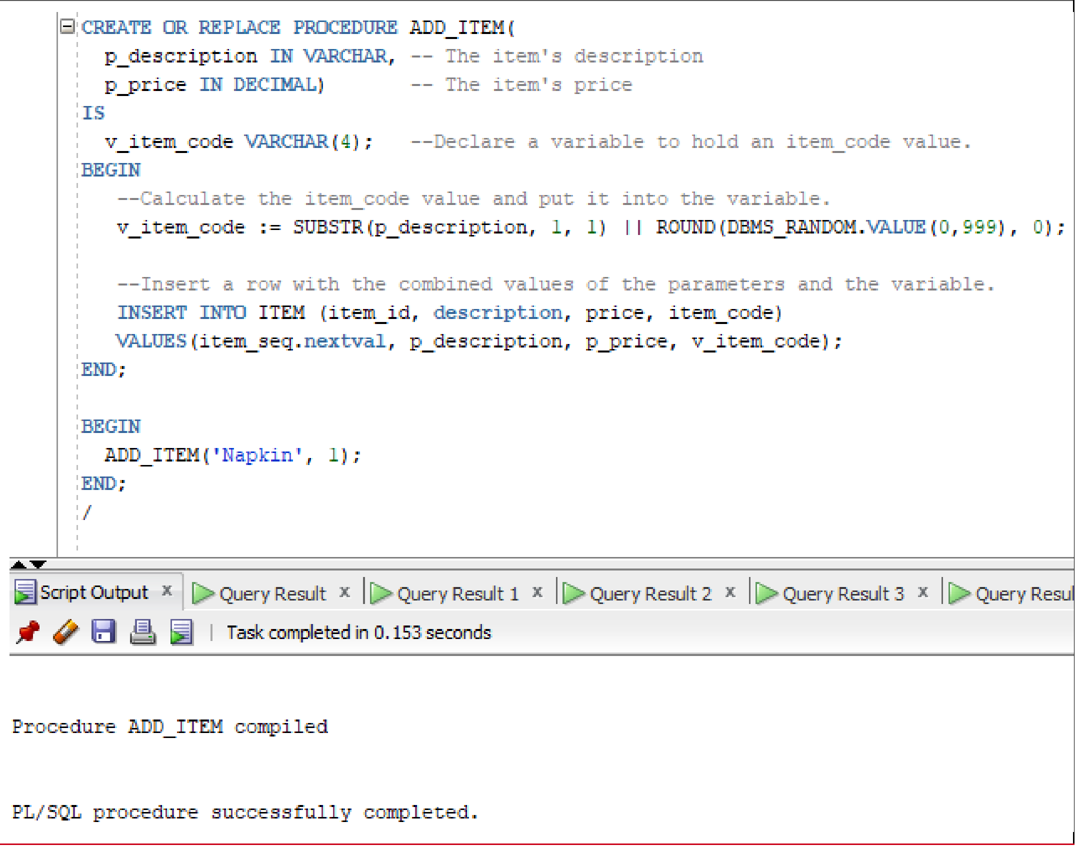
The SUBSTR function in Oracle returns a portion of a character string. The `SUBSTR(p_description, 1, 1)` code indicates to start at the first character (the first 1 argument), and to grab 1 character from there (the second 1 argument), thereby retrieving the first character. The DBMS_RANDOM.VALUE function obtains a random number between the given values (in this case, 0 to 999). The ROUND function rounds the number to the nearest whole number (since the 0 argument indicates no decimal points are desired). When the results of the SUBSTR function and the DBMS_RANDOM.VALUE are combined, it results in a 4-character item code as described above.

Last, you'll notice the insert line, which inserts the values into the Item table, with a reference to "v_item_code" for the item_code value. Referencing the variable instructs the SQL engine to pull the value stored in the variable.

Let's try out compiling and executing the stored procedure to add a "napkin" item.

Screenshot: Compiling and Executing add_item

Oracle

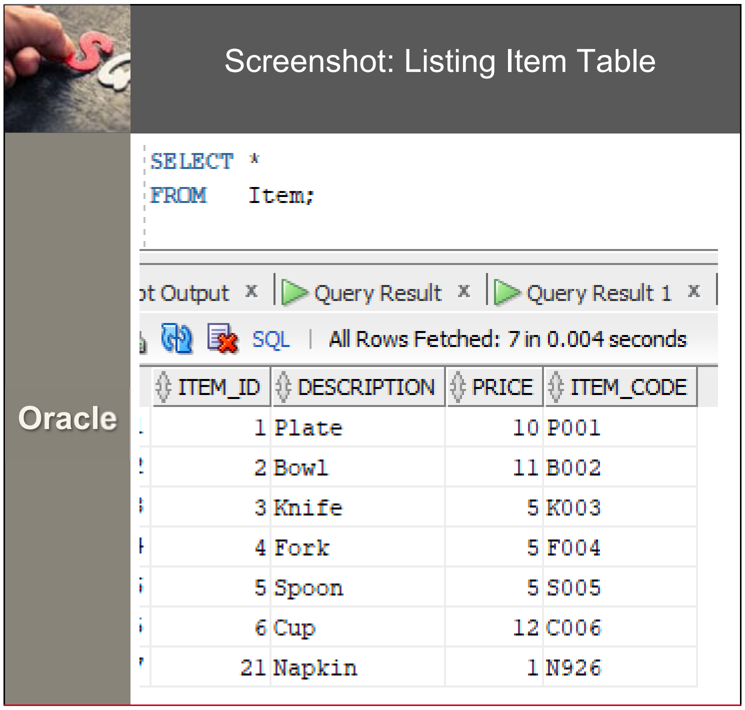


```
CREATE OR REPLACE PROCEDURE ADD_ITEM(
  p_description IN VARCHAR, -- The item's description
  p_price IN DECIMAL)       -- The item's price
IS
  v_item_code VARCHAR(4);   --Declare a variable to hold an item_code value.
BEGIN
   --Calculate the item_code value and put it into the variable.
   v_item_code := SUBSTR(p_description, 1, 1) || ROUND(DBMS_RANDOM.VALUE(0,999), 0);

   --Insert a row with the combined values of the parameters and the variable.
   INSERT INTO ITEM (item_id, description, price, item_code)
   VALUES(item_seq.nextval, p_description, p_price, v_item_code);
END;

BEGIN
  ADD_ITEM('Napkin', 1);
END;
/
```

Task completed in 0.153 seconds

```
Procedure ADD_ITEM compiled


PL/SQL procedure successfully completed.
```

Notice that we placed the “/” character after the stored procedure definition so that we could execute additional commands, in this case, the adding of the napkin item. We execute the add_item stored procedure just as we executed the add_customer procedure in the prior step. Let’s look at the item table now to see if our item was added.


Screenshot: Listing Item Table

Oracle

```
SELECT *
FROM   Item;
```

All Rows Fetched: 7 in 0.004 seconds

ITEM_ID	DESCRIPTION	PRICE	ITEM_CODE
1	Plate	10	P001
2	Bowl	11	B002
3	Knife	5	K003
4	Fork	5	F004
5	Spoon	5	S005
6	Cup	12	C006
21	Napkin	1	N926

Sure enough, the Napkin item was added, and the item_code value was automatically calculated rather than being passed in as a parameter by the executor. We achieved something new!

There is one more important concept you need to understand about variables to make effective use of them. *Variable scope* is the region in which a variable is accessible. In the examples in this step, we declare the variable within the stored procedure, which means that the variable is only accessible within that same stored procedure. Another stored procedure, or the SQL engine itself, cannot access the variable. When the procedure is invoked, the variable becomes accessible by code in the procedure. Unless a value is given in its declaration, the variable is initialized to null, and another line of code can explicitly set its value to something else.

What about multiple executions of the same procedure? Does the variable's value remain across executions? The simple answer is no. Every time the procedure is invoked, the variable's value is initialized and available only to that particular execution. Different executions of the stored procedure have access to different values of the variable. That is, even though it appears multiple executions are accessing the same variable since it carries the same and declaration, *each execution has its own copy of the variable* so that each execution can use the variable independent of another execution.

Hopefully the examples in this step help illustrate one purpose of using variables. A variable provides a place to store values, which can be calculated by using expressions, and then the variable can later be referenced to retrieve its value.

You now have enough knowledge to create the stored procedure for this step.

Step 6 – Create Lookup Procedure

What you're being asked to do is certainly becoming more complex, but don't worry, you already have most of the skills you need. You already know how to create parameterized stored procedures, and declare and use variables. The one skill you have not learned yet is setting the value of a variable based upon the results of a query. Since your procedure will be given a username and not a person_id, you will need to look this up by executing a query. There is a way to do this and store it into a variable.

We'll demonstrate how to do this by creating an "add_line_item" stored procedure that supports adding line items to the database. Rather than the executor specifying the item_id, the stored procedure will take the item_code as a parameter, then lookup the item_id. The other parameters will be specified as usual. Such a procedure can look like this.

Code: ADD_LINE_ITEM procedure

```
CREATE OR REPLACE PROCEDURE ADD_LINE_ITEM(
  p_item_code IN VARCHAR,   -- The code of the item.
  p_order_id IN DECIMAL,    -- The ID of the order for the line item.
  p_quantity IN DECIMAL)    -- The quantity of the item.
IS
  v_item_id DECIMAL(12);      --Declare a variable to hold the ID of the item code.
  v_line_price DECIMAL(12,2); --Declare a variable to calculate line price.
BEGIN
   --Get the item_id based upon the item_code, as well as the line total.
   SELECT item_id, price * p_quantity
   INTO   v_item_id, v_line_price
   FROM   Item
   WHERE  item_code = p_item_code;

   --Insert the new line item.
   INSERT INTO LINE_ITEM(item_id, order_id, item_quantity, line_price)
   VALUES(v_item_id, p_order_id, p_quantity, v_line_price);
END;
```

There are four columns in the line_item table – item_id, order_id, item_quantity, and line_price. Order_id and item_quantity are specified explicitly as parameters, so the executor decides what values to pass in explicitly. However, the other two are not specified explicitly, but are looked up by using the item_code. Notice that there are two variables declared, v_item_id and v_line_price; these will be used to store these values. It's this select statement that is interesting for this step.

```
SELECT item_id, price * p_quantity
INTO   v_item_id, v_line_price
FROM   Item
WHERE  item_code = p_item_code;
```

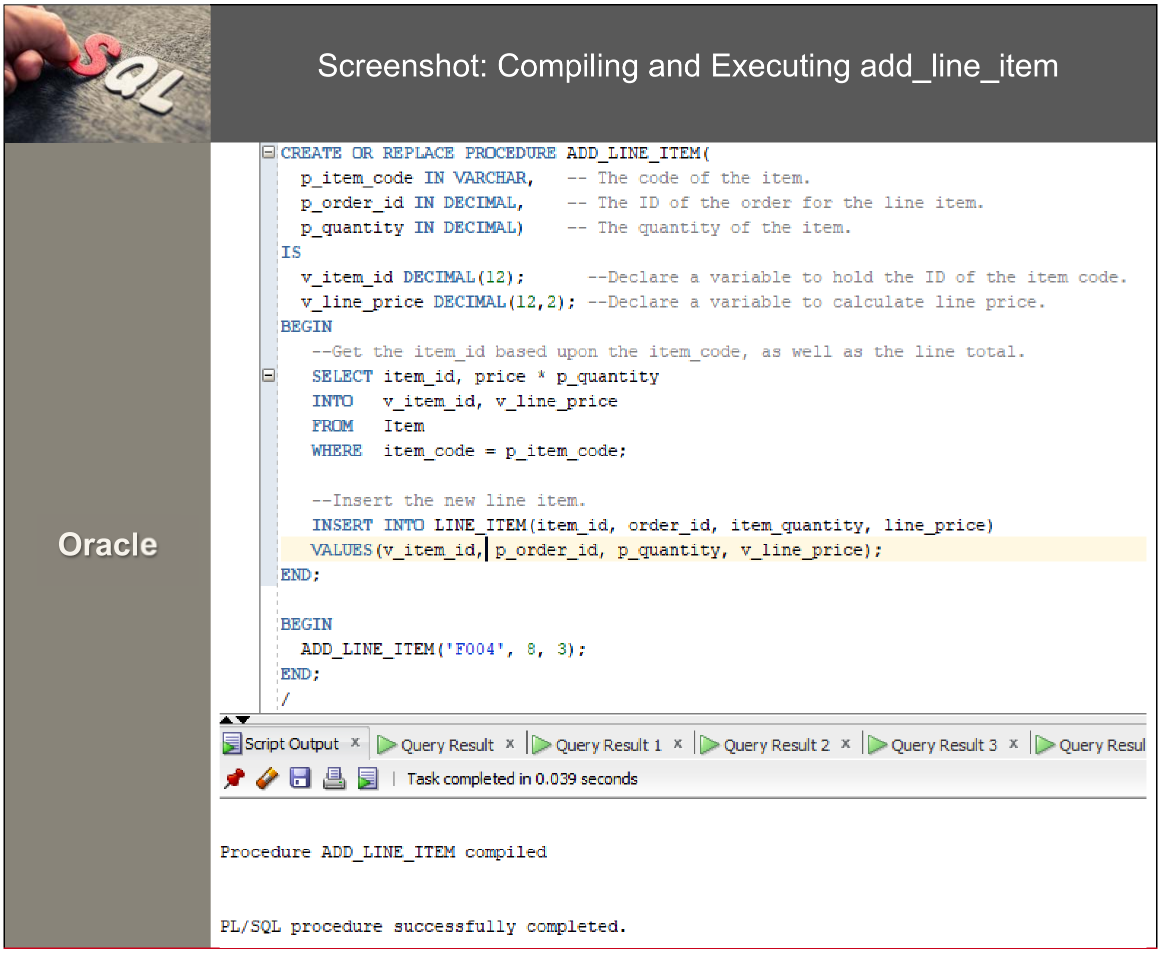
The standard SELECT, FROM, WHERE clauses combined are retrieving the item corresponding to the item code passed in as a parameter, and pulling back the item_id column, and calculating what the line price would be by multiplying the price times the quantity specified. You might have guessed that the clause we have not dealt with before, INTO (on the second line of the query), instructs the SQL engine to put the values retrieved into variables. The list of variables is correlated to the list of columns. That is, the item_id column corresponds to the v_item_id variable since they come first in both lists, and the price * quantity column corresponds to the v_line_price variable since they come second in both lists.

We refer to this statement as a whole as a SELECT INTO statement. After it is executed, both variables are populated with the values retrieved in the select. Do you see the power of the SELECT INTO statement? You can use it to lookup values in other tables, store them in variables, then later use those variables as needed. In our case, we are using the SELECT INTO to determine what the item_id and line_price values should be, then using those variables in our insert statement.

Next, let's use our stored procedure to add a line item for an order, where three "fork" items are added. As a reminder, the "fork" item has these values:

item_id	description	price	item_code
4	Fork	5	F004

It's item_code is "F004", its price is $5, and its ID is 4. Here is a screenshot of the code used to add three fork items to an order (order with id 8).


Screenshot: Compiling and Executing add_line_item

Oracle

```
CREATE OR REPLACE PROCEDURE ADD_LINE_ITEM(
  p_item_code IN VARCHAR,    -- The code of the item.
  p_order_id IN DECIMAL,     -- The ID of the order for the line item.
  p_quantity IN DECIMAL)     -- The quantity of the item.
IS
  v_item_id DECIMAL(12);      --Declare a variable to hold the ID of the item code.
  v_line_price DECIMAL(12,2); --Declare a variable to calculate line price.
BEGIN
   --Get the item_id based upon the item_code, as well as the line total.
   SELECT item_id, price * p_quantity
   INTO   v_item_id, v_line_price
   FROM   Item
   WHERE  item_code = p_item_code;

   --Insert the new line item.
   INSERT INTO LINE_ITEM(item_id, order_id, item_quantity, line_price)
   VALUES(v_item_id, p_order_id, p_quantity, v_line_price);
END;

BEGIN
  ADD_LINE_ITEM('F004', 8, 3);
END;
/
```

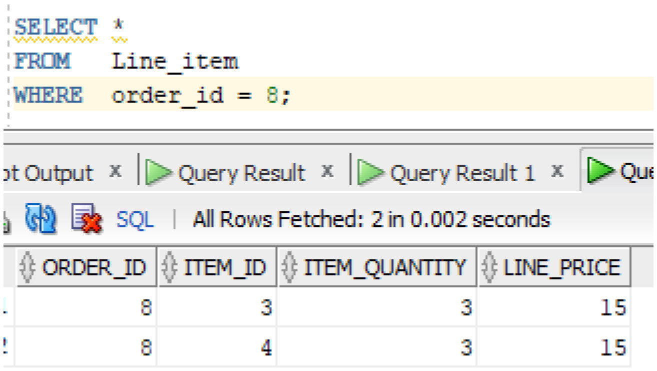
Task completed in 0.039 seconds

```
Procedure ADD_LINE_ITEM compiled


PL/SQL procedure successfully completed.
```

The ADD_LINE_ITEM procedure was invoked, referencing the "F004" item code, order id 8, and a quantity of 3. Now let's see if our results made it into the table by selecting all line items for order 8.

Screenshot: Listing Line_item After Execution

Oracle



```
SELECT *
FROM   Line_item
WHERE  order_id = 8;
```

All Rows Fetched: 2 in 0.002 seconds

ORDER_ID	ITEM_ID	ITEM_QUANTITY	LINE_PRICE
8	3	3	15
8	4	3	15

Notice that with ID 8 now has an additional line item for forks (with ID 4), quantity 3, and a price of $15 (since $5 * 3 = $15).

We are able to use this procedure to automatically calculate the line price, and to retrieve the correct item, all with its item code. Do you see now the power of lookups and storing the values in variables using the SELECT INTO statement? It gives a new dimension to your stored procedures, as your procedures can now take parameters, lookup values in various tables, calculate values, and use them as needed. You're on your way to becoming an expert!

Now you can use a similar strategy to create your procedure.

Step 7 – Single Table Validation Trigger

To demonstrate how to do this, we will create a trigger that prevents the customer balance from being negative. This validation only needs information from the table being modified and so qualifies as an intra-table validation. Below is the code for such a trigger.

Code: No Negative Balance Validation Trigger

```
CREATE OR REPLACE TRIGGER no_neg_bal_trg
BEFORE UPDATE OR INSERT ON Customer
FOR EACH ROW
BEGIN
   IF :NEW.customer_total < 0 THEN
   RAISE_APPLICATION_ERROR(-20001,'Customer balance cannot be negative.');
   END IF;
END;
```

Since we have not yet reviewed triggers, let's examine this line by line.

`Line 1: CREATE OR REPLACE TRIGGER no_neg_bal_trg`

The **`CREATE OR REPLACE TRIGGER`** phrase indicates to Oracle that a trigger is to be created, and that the creation replaces any existing definition of the trigger. If we instead use the phrase **`CREATE TRIGGER,`** then Oracle would create the trigger only if one with the same name does not exist.

The **`no_neg_bal_trg`** word is the name of the trigger, an identifier of our own choosing. We put "trg" at the end of the identifier as a convention, so it's recognizable as the name of a trigger. The rest of the name helps describe what the trigger does, which is validate that there is no negative balance.

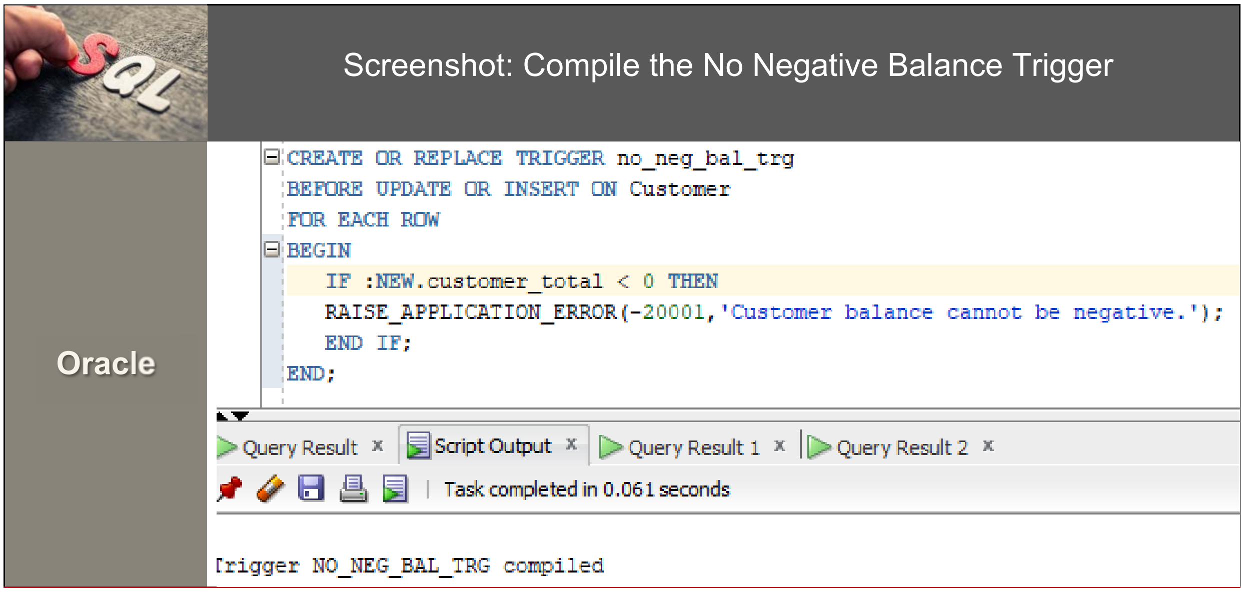
`Line 2: BEFORE UPDATE OR INSERT ON Customer`

`BEFORE` is a SQL keyword that instructs Oracle the trigger is to be executed before the insert or update occurs in the database. **`UPDATE OR INSERT`** indicates that the trigger is to be fired when either an insert or an update statement happens on the table. If we had omitted "INSERT" for example, then the trigger would only fire when an update occurs. We want to block all negative balances so we have the trigger fire on both updates and inserts. **`ON Customer`** ties to trigger to the Customer table, that is, indicates to Oracle that the aforementioned actions (update and insert) will fire this trigger only if they happen on the Customer table. Triggers are inexorably linked to one table by definition. If the table is dropped, the trigger is also automatically dropped.

`Line 3: FOR EACH ROW`

	This tells Oracle that the trigger is a *row-level trigger*, which means it is executed *once for each row* that is affected by the triggering SQL statement. For example, an UPDATE statement In Oracle can affect many rows in a table, and the **FOR EACH ROW** words ensure the trigger is executed once for each affected row. Only row-level triggers can access the before and after column values, and since we want to use this trigger for validation, we must make it a row-level trigger. If these words were omitted, the trigger would be a statement-level trigger, which would mean it would execute only once regardless of the number of rows affected (and wouldn't have access to each row's before and after values).
Line 4: BEGIN	
	Just as with stored procedures and anonymous PL/SQL blocks, code for triggers needs to be defined within a BEGIN/END block. The BEGIN keyword opens the block for the trigger.
Line 5: IF :NEW.customer_total < 0 THEN	
	The **IF/THEN** keywords tell Oracle that the block following is only to be executed of the Boolean expression evaluates to true. If statements allow us to conditionally execute code. For this trigger, we only want to reject SQL statements that attempt to create a negative balance, so we use an if statement. The **:NEW.customer_total < 0** component is the Boolean expression that the if statement will evaluate. :NEW is a pseudo-table that has all of the same columns as the table the trigger is attached to (in this case, the Customer table), and a single row populated with the values of the row being modified, after they have been modified. If we wanted to look at the values before they have been modified, we could use the :OLD pseudo-table. The Boolean thus evaluates to true only when the new customer_total value is less than 0 (negative).
Line 6: RAISE_APPLICATION_ERROR(-20001,'Customer balance cannot be negative.');	
	The library routine RAISE_APPLICATION_ERROR is provided by Oracle so that custom errors can be raised when needed. The routine make provision for a human-readable error message and an ORA error code, and rolls back the transaction with the error, "Customer balance cannot be negative". We provided the -20001 as the first argument to RAISE_APPLICATION_ERROR, which will translate to an ORA error number (cross-reference the screenshot below).
Line 7: END IF;	
	This ends the IF statement block, so that Oracle knows which code is conditional upon the if statement, and which code is unconditionally executed.
Line 8: END;	
	This ends the block for the trigger and is also the last part of the trigger definition.

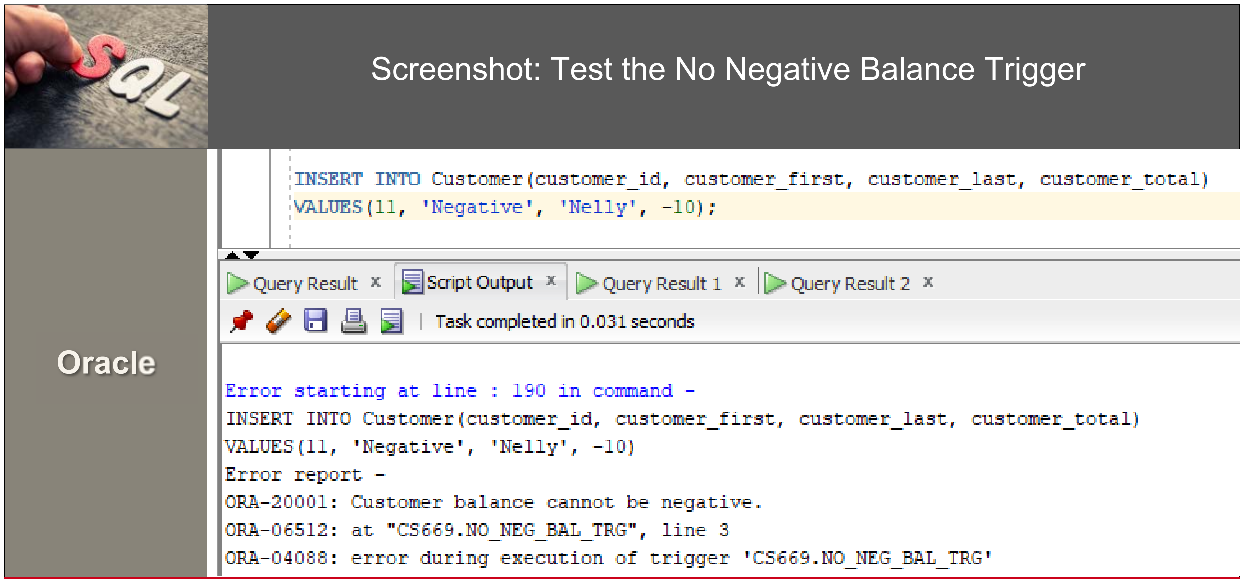
First, we compile the trigger as illustrated in the screenshot below.


Screenshot: Compile the No Negative Balance Trigger
Oracle
Query Result
Script Output
Query Result 1
Query Result 2
Task completed in 0.061 seconds

```
CREATE OR REPLACE TRIGGER no_neg_bal_trg
BEFORE UPDATE OR INSERT ON Customer
FOR EACH ROW
BEGIN
   IF :NEW.customer_total < 0 THEN
   RAISE_APPLICATION_ERROR(-20001,'Customer balance cannot be negative.');
   END IF;
END;
```

```
Trigger NO_NEG_BAL_TRG compiled
```

As soon as the trigger is compiled successfully, it is active in the database and will execute when the triggering event happens from that point forward (until, of course, the trigger is disabled or dropped). If we attempt to insert a customer with a negative balance, the trigger will fire and reject it, shown below.


Screenshot: Test the No Negative Balance Trigger
Oracle
Query Result
Script Output
Query Result 1
Query Result 2
Task completed in 0.031 seconds

```
INSERT INTO Customer(customer_id, customer_first, customer_last, customer_total)
VALUES(11, 'Negative', 'Nelly', -10);
```

```
Error starting at line : 190 in command -
INSERT INTO Customer(customer_id, customer_first, customer_last, customer_total)
VALUES(11, 'Negative', 'Nelly', -10)
Error report -
ORA-20001: Customer balance cannot be negative.
ORA-06512: at "CS669.NO_NEG_BAL_TRG", line 3
ORA-04088: error during execution of trigger 'CS669.NO_NEG_BAL_TRG'
```

Notice that the insert was immediately rejected by the trigger, that the error code reported back is ORA-20001 (coming from our -20001 value), and the message is “Customer balance cannot be negative.” We cannot execute the trigger directly, but we can see the effects of the trigger when we execute a triggering statement such as an insert.

You can use similar logic to create your validation trigger.

Step 8 – Cross-Table Validation Trigger

To demonstrate cross-table validation, imagine we want to validate the fact that the line price for a line item actually equals the quantity times the item price. The quantity is stored in the Line_item table while the item price is stored in the Item table. We can setup a trigger on the Line_item table to perform this validation using constructs we've already used in prior steps in this lab. Here is the code for such a trigger.

Code: Correct Line Price Validation Trigger

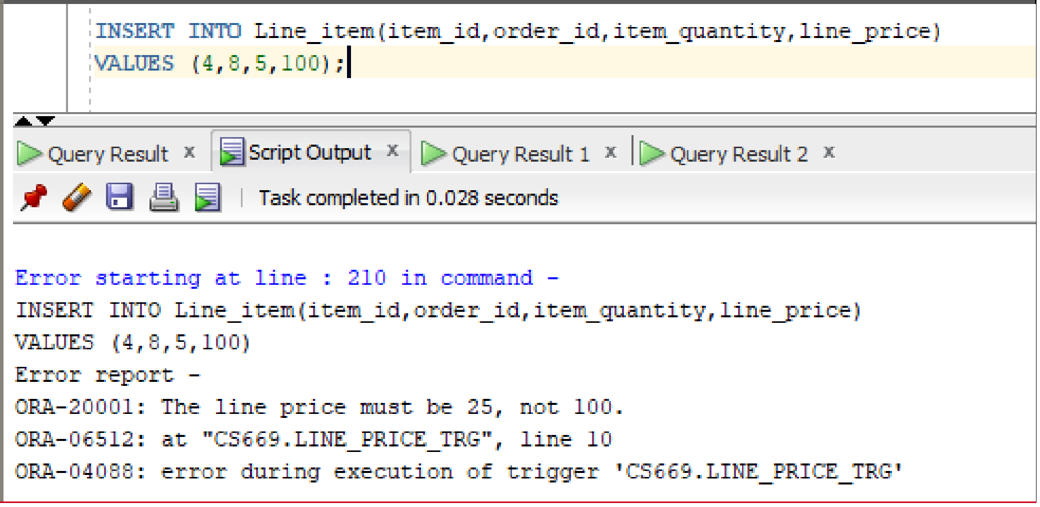
```
CREATE OR REPLACE TRIGGER line_price_trg
BEFORE UPDATE OR INSERT ON Line_item
FOR EACH ROW
DECLARE
  v_correct_line_price DECIMAL(12,2);
BEGIN
   SELECT :NEW.item_quantity * Item.price
   INTO v_correct_line_price
   FROM   Item
   WHERE  item.item_id = :NEW.item_id;

   IF :NEW.line_price <> v_correct_line_price THEN
     RAISE_APPLICATION_ERROR(-20001,'The line price must be ' || v_correct_line_price ||
     ', not ' || :NEW.line_price || '.');
   END IF;
END;
```

You've seen all of the constructs here individually, but the integration of them needs more explanation. We put a DECLARE block after the FOR EACH ROW statement so that we could declare a variable that will store the correct line price. Then the first line of the body of the trigger uses a query to multiply the item quantity times the item price, and stores it into the v_correct_line_price variable. It then uses an if statement to determine if the line price of the new or updated row is correct. If it's not, it raises an application error that indicates what the line price is supposed to be. You've seen all of these constructs before, so you're not witnessing just another use case.

Let's try it out. We'll try to insert a line item with an invalid line price, as follows.

Screenshot: Test the Price Validation Trigger

Oracle



```
INSERT INTO Line_item(item_id,order_id,item_quantity,line_price)
VALUES (4,8,5,100);
```

Task completed in 0.028 seconds

```
Error starting at line : 210 in command -
INSERT INTO Line_item(item_id,order_id,item_quantity,line_price)
VALUES (4,8,5,100)
Error report -
ORA-20001: The line price must be 25, not 100.
ORA-06512: at "CS669.LINE_PRICE_TRG", line 10
ORA-04088: error during execution of trigger 'CS669.LINE_PRICE_TRG'
```

We attempted to insert a line item with a line price of 100, and the trigger rejected it because the line price should be 25. Why? Item with ID 4 has a price of 5, and there is a quantity of 5, so the line price would be 25 and not 100. We successfully used a trigger to perform a cross-table validation, simply by combining constructs we were already familiar with.

With all of these skills, you may begin feel very powerful. Just make sure to use your powers for good and not evil. You can use similar logic to create your cross-validation trigger.

Step 9 – History Trigger

To demonstrate capturing history, imagine that we would like to store a history of price changes for each item, so that we know the price of an item at any point in time despite any price updates. Abstractly, these fields should be included in a history table – a reference (foreign key) to the table being changed, the old value of the column, the new value of the column, and the date of change. You can think of this set of fields as a design pattern for history tables. For our example, we would create an Item_price_history table that stores a reference to the item, its old price, its new price, and the date of the change.

Before showing you code, let's make sure to differentiate history and auditing, which have two different purposes. A history table records changes over time but remains active in the schema, with proper foreign keys for references. The fields are setup so that SQL queries and transactions can use the table along with the other tables in the schema, that is, so that the table can be used regularly like any other table in the schema. The purpose of a history table is to make prior values available to the people and applications that use the database in a way that coexists with the current values.

An audit table also records changes over time, but it does not remain active in the schema. The purpose of an audit table is to simply record that a change happened in a way that people can manually review the changes later in case of any concern or dispute. An audit table has no foreign keys, and acts more like a log which contains the full value of each field. Since an audit table does not make use of foreign keys, and flattens out the needed fields, it survives schema changes over time well. For example, as already mentioned a history table for price changes would have a foreign key to the item, but an audit table would instead have the description of the item along with any other information needed to identify the item. Someone could manually review the audit table to see which prices changes over time for which items.

It's a best practice to be aware of which kind of table you're creating – history or audit – and follow the design patterns for that table. Some organizations inadvertently create hybrid tables that perhaps start out strictly for auditing, but then later add in foreign keys, and this can cause problems as changes are made to the database. In this step, we are creating a history table that remains active in the schema.

First, let's look at the code for creating the Item_price_history table, below.

Code: Create the Item_price_history Table

```
CREATE TABLE Item_price_history (
item_id DECIMAL(12) NOT NULL,
old_price DECIMAL(10,2) NOT NULL,
new_price DECIMAL(10,2) NOT NULL,
change_date DATE NOT NULL,
FOREIGN KEY (item_id) REFERENCES Item(item_id));
```

You're very familiar with table creation syntax at this point, so there's no need to belabor the basics of this SQL. Instead, we'll focus on what's important here. We opted to avoid giving this table a primary key; a primary key is not necessary to record the history. In some real-world schemas, the standards employed by the organization may mandate a primary key, especially if the schema will be mapped by an object-relational mapping tool within an application. There is a foreign key to the Item which has been changed. The old_price and new_price columns have the same datatype as the original Item.price column, since it will be a record of the change.

Once the table is created, we then define a trigger on the Item table that inserts a row into the Item_price_history table whenever an item price is updated. The code for the trigger is below.

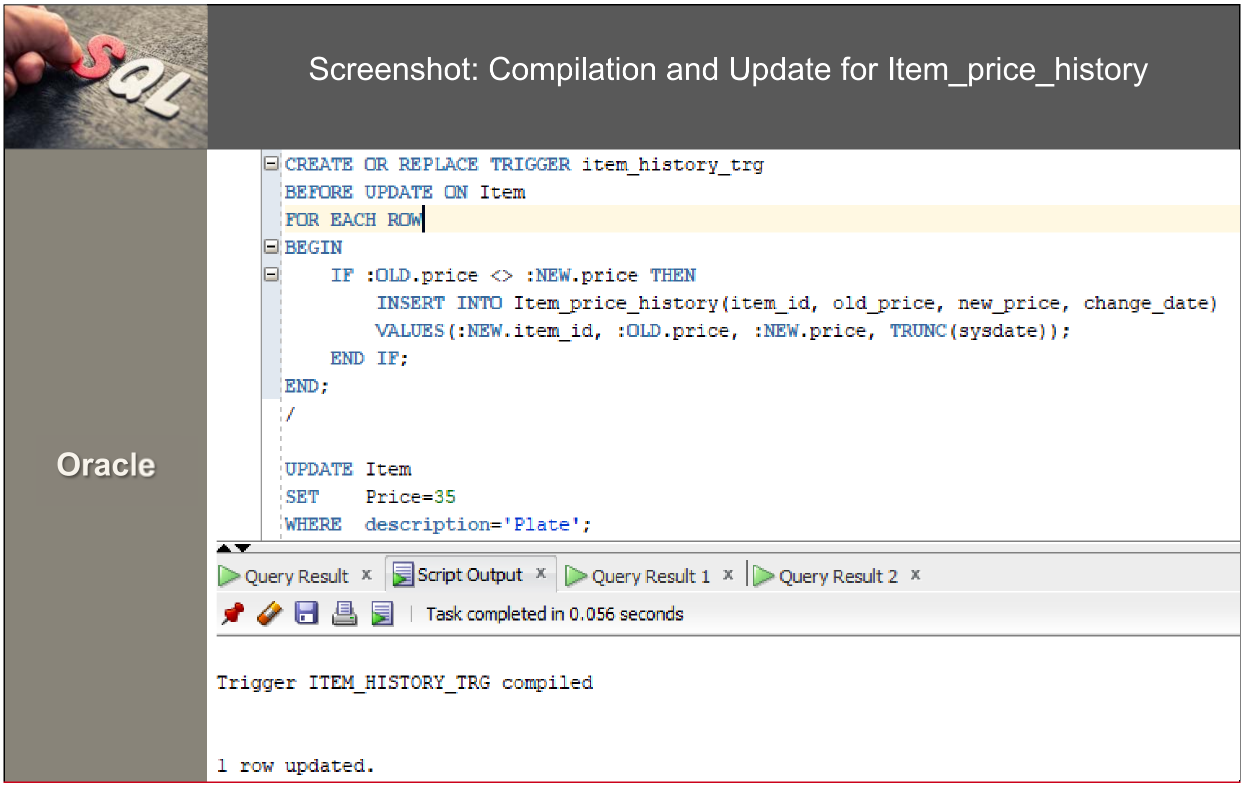
Code: The Item_price_history Trigger

```
CREATE OR REPLACE TRIGGER item_history_trg
BEFORE UPDATE ON Item
FOR EACH ROW
BEGIN
    IF :OLD.price <> :NEW.price THEN
        INSERT INTO Item_price_history(item_id, old_price, new_price, change_date)
        VALUES(:NEW.item_id, :OLD.price, :NEW.price, TRUNC(sysdate));
    END IF;
END;
```

This trigger only fires on updates, because we only want to record changes in price. We've opted not record the initial price so we don't have the trigger fire on insert; however, one variation of the history table would record every price including the initial one. In that case, the trigger would be modified to also trigger on insert, and the old_price column would be nullable so that when the first price is created, the old_price is null (since there is no price).

The trigger only records an update if the old price is different than the new price, so an if statement is used. The :OLD pseudo-table is used to retrieve the old price, and the :NEW pseudo-table is used to retrieve the new price and the item ID. The keyword "sysdate" is used to retrieve the current date and time, and the TRUNC function eliminates the time portion of the value, leaving only the date portion.

Let's test that our trigger works by modifying the price of an item in the Item table. The compilation and update are illustrated first, below.



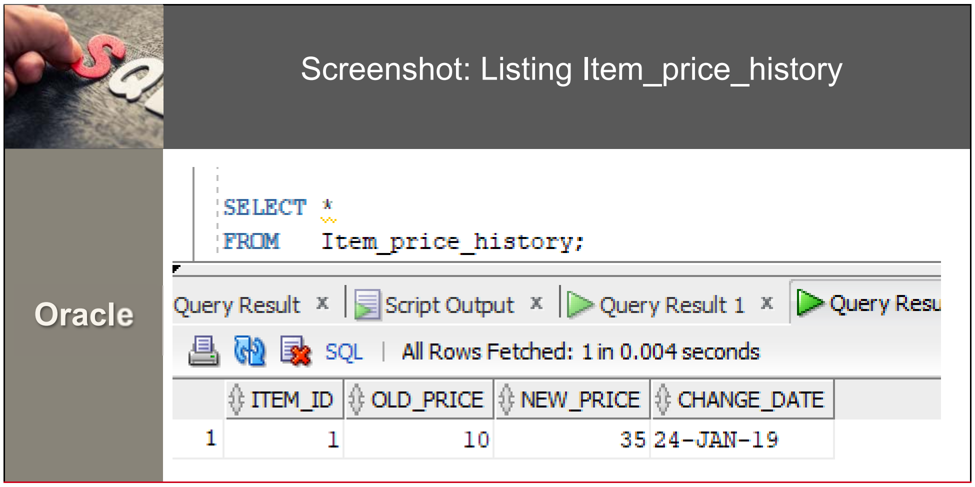
Screenshot: Compilation and Update for Item_price_history

```
CREATE OR REPLACE TRIGGER item_history_trg
BEFORE UPDATE ON Item
FOR EACH ROW
BEGIN
    IF :OLD.price <> :NEW.price THEN
        INSERT INTO Item_price_history(item_id, old_price, new_price, change_date)
        VALUES(:NEW.item_id, :OLD.price, :NEW.price, TRUNC(sysdate));
    END IF;
END;
/

UPDATE Item
SET    Price=35
WHERE  description='Plate';
```

Task completed in 0.056 seconds

```
Trigger ITEM_HISTORY_TRG compiled


1 row updated.
```

Next, the listing of the table itself is included, to show that the trigger recorded the update.



Screenshot: Listing Item_price_history

```
SELECT *
FROM   Item_price_history;
```

All Rows Fetched: 1 in 0.004 seconds

	ITEM_ID	OLD_PRICE	NEW_PRICE	CHANGE_DATE
1	1	10	35	24-JAN-19

We now see a row in the history table indicating the item had an old price of 10, a new price of 35, and the change happened on the specific date. With this structure, all such price changes will be recorded over time. And following this pattern, we could record a history for whatever column we like for whatever table we need. Amazing!

You can follow this pattern to create and populate the history table for your lab.

Step 10 – Creating Normalized Table Structure

The scope and technical complexity on how to perform normalization from scratch is too broad and deep to be a part of this document. Please use the textbook, online lectures, and live classroom sessions to learn about normalization, how to identify and represent functional dependencies, and the steps to follow to normalize a table. Keep in mind that it is best practice to normalize tables to BCNF when possible. If a table is not normalized to BCNF for specific reasons, we should be aware of those reasons and make a conscious choice to do so.