



## Code Pods – **FREE** SAMPLES

Thank you for downloading this document which contains twelve free samples from our ever popular **code pods** series. Bearing in mind that the real code pods have a total of:

- Over **900+** pages of detailed explanation, annotation and results
- Around **1000** SQL code samples – all fully tested and annotated

Then you can appreciate that these free samples represent just a *very small fraction* of the material available. However, they do give you a flavour of the real code pods. Full details of the code pods can be seen at:

[www.code-stream.co.uk](http://www.code-stream.co.uk)

If you like what you see here then please visit the **Code Stream** website and have a look in more detail at how we can help you write better SQL code. Full details of each code pod and how to purchase them are available on that website. You can also see the **Special Offers** available.

Alternatively, please email us directly on:

[feedback@knowledge-river.co.uk](mailto:feedback@knowledge-river.co.uk)

for general help and advice – without any obligation. We would love to hear from you.

The free samples begin over the page...

## Free Sample 1: Table creation with constraints

```
CREATE TABLE Student
(StudentID  INTEGER          DEFAULT '9999999' NOT NULL UNIQUE,
 Title     CHAR (4)         CHECK (Title IN ('Mr', 'Mrs', 'Miss')),
 FirstName VARCHAR(15)     NOT NULL,
 Initials  VARCHAR(3)      CHECK (Initials IN '[A-Z]'),
 LastName  VARCHAR(20)     NOT NULL,
 Degree    VARCHAR(4)      CHECK (Degree IN
 ('BSc', 'BEng', 'MSc', 'MEng', 'PhD')),
 Subject   VARCHAR (50)    CHECK (Subject IN
 ('Computing', 'Maths', 'Electronics')),
 Year      INTEGER         CHECK (Year BETWEEN 1 AND 4),
 DoB       DATE            DEFAULT '01-JAN-2008' NOT NULL,
 Address   VARCHAR(100)    DEFAULT 'The University',
 Email     VARCHAR (30)    DEFAULT 'post@learnwell.ac.uk',
 Telephone VARCHAR (12)    CHECK (Telephone IN '[0-9]'),
 Sex       CHAR(1)         NOT NULL
                        CHECK (Sex IN ('M', 'm', 'F', 'f')),
 Fees      DECIMAL(7,2)    DEFAULT '3000.00'
                        CHECK (Fees > 0.00),
 Debt      DECIMAL(7,2)    DEFAULT '00000.00'
                        CHECK (Debt > 0.00));
```

Table created.

## Free Sample 2: Naming CHECK Constraints

```
CREATE TABLE Module
(ModuleCode  VARCHAR(4),
 ModuleTitle VARCHAR(40),
 ModuleDesc  VARCHAR(500),
 ModuleCredits  INTEGER,
 ModuleDuration  INTEGER,
 ModuleStartYear  INTEGER,
 ModuleEndYear   INTEGER,
 CONSTRAINT Mod_PK          PRIMARY KEY (ModuleCode),
 CONSTRAINT Mod_Size        CHECK (ModuleCredits IN (15, 30, 60)),
 CONSTRAINT Mod_Length      CHECK (ModuleDuration BETWEEN 3 AND 12),
 CONSTRAINT Mod_Begin       CHECK (ModuleStartYear > 2008),
 CONSTRAINT Mod_End         CHECK (ModuleEndYear < 2012));
```

Table created.

### Note:

Just like primary key and foreign key constraints, each CHECK constraint, once given its own unique identifier, can be tracked and traced through the DBMS data dictionary. Get into the habit of naming all your table constraints – then you can find them quickly when searching through the meta-data in the dictionary. The precise code you need to enter to do such dictionary searches is DBMS-dependent – so check with your local documentation – but it will look *something* similar to...

```
SELECT TableName, ConstraintName, ConstraintType
FROM UserTables
ORDER BY TableName;
```

(The precise names of the columns and tables will depend on the DBMS)

**Free Sample 3: Using GROUP BY and HAVING**Points to note:

The HAVING clause acts as *group-level* filter (just as WHERE acts as a *row-level* filter). In this example, only those groups (product types) that have an average cost over £250 will be displayed.

```
SELECT ProductName, AVG (Cost) As AveragePrice,
       MIN (Cost) AS Cheapest, MAX (Cost) AS Dearest
FROM QueryDemo
GROUP BY ProductName
HAVING AVG(Cost) > 250 AND MIN(Cost) > 200;
```

PRODUCTNAME	AVERAGEPRICE	CHEAPEST	DEAREST
Laptop	599.99	599.99	599.99

**Free Sample 4: Using SUB-QUERIES**

```
SELECT ProductID, ProductName, Description
FROM QueryDemo
WHERE Cost > (SELECT AVG (Cost) FROM QueryDemo
             WHERE Supplier IN
             (SELECT Supplier FROM QueryDemo
             WHERE Stocklevel < 50));
```

PRODUCTID	PRODUCTNAME	DESCRIPTION
1	TV	42 inch widescreen. HD ready
2	TV	50 inch surround sound plasma
4	TV	60 inch HD-ready LCD
3	Laptop	17 inch widescreen

## Free Sample 5: Using SET Operations

```
SELECT ProductID, ProductName, Description
FROM QueryDemo
WHERE Cost > 500
```

**INTERSECT**

```
SELECT ProductID, ProductName, Description
FROM QueryDemo
WHERE StockLevel < 100;
```

PRODUCTID	PRODUCTNAME	DESCRIPTION
1	TV	42 inch widescreen. HD ready
2	TV	50 inch surround sound plasma
3	Laptop	17 inch widescreen
4	TV	60 inch HD-ready LCD

**4 rows selected.**

By changing the second clause we get...

```
SELECT ProductID, ProductName, Description
FROM QueryDemo
WHERE Cost > 500
```

**INTERSECT**

```
SELECT ProductID, ProductName, Description
FROM QueryDemo
WHERE StockLevel < 20;
```

PRODUCTID	PRODUCTNAME	DESCRIPTION
2	TV	50 inch surround sound plasma
4	TV	60 inch HD-ready LCD

**2 rows selected.**

**Free Sample 6: Using multi-part WHERE clauses**

```
SELECT ProductID, ProductName, Supplier, Cost
FROM QueryDemo
WHERE Cost BETWEEN 1000 AND 2000
AND ProductName IN ('TV', 'Mobile', 'DVD')
OR NOT (Supplier IN ('Palm Power', 'One World', 'On Your Way'));
```

PRODUCTID	PRODUCTNAME	SUPPLIER	COST
1	TV	TVs4U	999.99
2	TV	TVs4U	1200
4	TV	TVsDirect	2000
5	DVD	Vision Aces	129.99
6	DVD	Vision Aces	139.99
11	TV	Micro Vision	99.99
12	MP3	Music on the Move	79.99
13	MP3	Music on the Move	129
14	MP3	Massive Music	200
18	Mobile	Tiny Talk	20
19	Mobile	Tiny Talk	50
20	Mobile	Big Talk	129
3	Laptop	In Your Lap	599.99

13 rows selected.

**Free Sample 7: Using String Concatenation & Expressions**

```
SELECT 'Product ID ' || ProductID || ' (' || ProductName || ') ' ||
'costs £' || Cost*1.175 || ' (inc. VAT) '
FROM QueryDemo;
```

Produces...

```
Product ID 1 (TV) costs £1174.98825 (inc. VAT)
Product ID 2 (TV) costs £1410 (inc. VAT)
Product ID 4 (TV) costs £2350 (inc. VAT)
Product ID 5 (DVD) costs £152.73825 (inc. VAT)
Product ID 6 (DVD) costs £164.48825 (inc. VAT)
Product ID 7 (SatNav) costs £176.25 (inc. VAT)
Product ID 8 (SatNav) costs £129.25 (inc. VAT)
Product ID 9 (SatNav) costs £293.75 (inc. VAT)
Product ID 10 (SatNav) costs £152.75 (inc. VAT)
Product ID 11 (TV) costs £117.48825 (inc. VAT)
Product ID 12 (MP3) costs £93.98825 (inc. VAT)
Product ID 13 (MP3) costs £151.575 (inc. VAT)
Product ID 14 (MP3) costs £235 (inc. VAT)
Product ID 15 (PDA) costs £233.825 (inc. VAT)
Product ID 16 (PDA) costs £351.325 (inc. VAT)
Product ID 17 (PDA) costs £264.375 (inc. VAT)
Product ID 18 (Mobile) costs £23.5 (inc. VAT)
Product ID 19 (Mobile) costs £58.75 (inc. VAT)
Product ID 20 (Mobile) costs £151.575 (inc. VAT)
Product ID 3 (Laptop) costs £704.98825 (inc. VAT)
```

20 rows selected.

## Free Sample 8: Using the LIKE Operator & Wildcards

```
SELECT ProductID, ProductName, Cost, Supplier
FROM QueryDemo
WHERE Supplier LIKE 'T_ny%';
```

PRODUCTID	PRODUCTNAME	COST	SUPPLIER
18	Mobile	20	Tiny Talk
19	Mobile	50	Tiny Talk

### Key Points:

Only those rows holding a supplier name:

- Beginning with 'T'
- AND
- Containing the phrase 'ny'
- AND
- Having a *single* character in the middle (due to the underscore)
- AND
- Having zero or more characters after 'ny' (due to the % symbol)

will be selected!

## Free Sample 9: Creating a (Stored) Procedure with Parameters

```
CREATE PROCEDURE FlexiFiller
(p_ID      IN INTEGER,
 p_Name    IN VARCHAR(20),
 p_Price   IN DECIMAL(6,2)) AS
BEGIN
  INSERT INTO Routines VALUES (p_ID, p_Name, p_Price);
  COMMIT;
END;
/
```

**Warning: Procedure created with compilation errors.**

### Key Point:

Do not specify the SIZE of the parameters – only the DATA TYPE.

This problem is resolved by removing the old procedure and re-building it...

```
DROP PROCEDURE FlexiFiller;
```

**Procedure dropped.**

```
CREATE PROCEDURE FlexiFiller
(p_ID      IN INTEGER,
 p_Name    IN VARCHAR,
 p_Price   IN DECIMAL) AS
BEGIN
  INSERT INTO Routines VALUES (p_ID, p_Name, p_Price);
  COMMIT;
END;
/
```

**Procedure created.**

Notice that this procedure has THREE *formal* parameters (in bold) – all input parameters.

We can now invoke (call) this procedure...

```
BEGIN
  FlexiFiller(13, 'Cooker', 350.00);
END;
```

**Procedure successfully completed.**

Here the 'real' data values 13, Cooker and 350.00 are the *actual* parameters (arguments).

Each actual parameter is mapped across to the corresponding formal parameter in the procedure (in a strict place-for-place manner – so first actual goes to first formal, second actual to second formal etc – so get the data values in the correct order).

You can see it worked overleaf...

## Code Pods - FREE SAMPLES

```
SELECT * FROM Routines;
```

ID	NAME	PRICE
1	Toaster	24.75
2	Television	425
3	Radio	16.5
10	Computer	600
5	MP3 Player	38.5
6	DVD	160
7	DVD	185
8	Freezer	220
9	Fridge	220
11	Computer	750
12	Microwave	130
13	Cooker	350

**12 rows selected.**

**Free Sample 10: Creating a (Row-Level) Trigger**

```
SELECT * FROM Stock;
```

ID	DESCRIPTION	PRICE	STOCKLEVEL
1000	DVD	99.99	179
1001	Television	189.99	119
1004	Laptop	599.99	52
1003	Freezer	439.99	35

```
CREATE TRIGGER InsertingRows
  AFTER INSERT ON Stock
  FOR EACH ROW
  DECLARE
    v_TempRecord Stock%ROWTYPE; // local variable based on table row
  BEGIN
    v_TempRecord.ID := :new.ID;
    v_TempRecord.Description := :new.Description;
    DBMS_OUTPUT.PUT_LINE('New stock entered with ID: ' ||
      v_TempRecord.ID);
    DBMS_OUTPUT.PUT_LINE('This has product description: ' ||
      v_TempRecord.Description);
  END;
/
```

**Trigger created.**

Key Points:

The variable 'v\_TempRecord' only exists inside the trigger (it is a local variable – just like those used in procedures and functions) but it has special properties:

1. The clause %ROWTYPE tells the Oracle DBMS that **v\_TempRecord is a record of the exact same format as one complete row from the Stock table.**
2. It holds the currently processed row.
3. The old and new versions of this (modified) row can be accessed using the special keywords **:old** and **:new** (and these can then be copied into other local variables for transfer to other tables or simply displayed to screen).

The above syntax is Oracle-specific BUT the same concept is applicable across most DBMS – we show alternate syntax soon. We can now do a sample insertion and test the trigger...

```
INSERT INTO Stock VALUES (1005, 'Electric Fan', 21.99, 44);
```

```
ALERT! Someone has changed the STOCK table!
The Stock is about to be modified
New stock entered with ID: 1005
This has product description: Electric Fan
The Stock has been modified
```

**1 row created.**

```
// So the trigger worked fine.
```

**Free Sample 11: Application to extract data from a table**Key Points:

Here we are going to run queries against sample tables and 'catch' the returned data in variables that we have already set up. Just remember one very important point. If the query returns a *single* row of data then normal variables are fine BUT if *two or more* rows are returned we have a problem (because a variable can only hold *one value at a time*). In such cases, the variables would be overwhelmed by data and could not cope. This situation can only be resolved by using **CURSORS** – which we cover in another code pod.

That is why all of these examples only extract a *single* row of data into the waiting variables.

Let's begin by checking the current contents of our sample table...

```
SELECT * FROM test1;
```

COL1	COL2	COL3
1	This is row 1	01-SEP-08
2	This is row 2	02-SEP-08
3	This is row 3	03-SEP-08
4	This is row 4	04-SEP-08

```
DECLARE                                // We have 3 columns so need 3 variables
  var1  INTEGER;                        // Vars must be same data type as columns
  var2  VARCHAR(20);
  var3  DATE;
BEGIN
  SELECT coll, col2, col3              // Extract three column values
  INTO var1, var2, var3                // into three waiting variables
  FROM test1
  WHERE coll = 3;                      // Ensure we extract only ONE row

  DBMS_OUTPUT.PUT_LINE
    ('You have extracted the following row of data:');
  DBMS_OUTPUT.PUT_LINE(var1);
  DBMS_OUTPUT.PUT_LINE(var2);
  DBMS_OUTPUT.PUT_LINE(var3);        // Output variables to screen
END;
/
```

You have extracted the following row of data:

```
3
This is row 3
03-SEP-08
```

Procedure successfully completed.

## Code Pods - FREE SAMPLES

This extended application extracts **two** rows *at different times*

```
DECLARE
  var1  INTEGER;
  var2  VARCHAR(20);
  var3  DATE;
BEGIN
  SELECT col1, col2, col3      // First extraction into variables
  INTO var1, var2, var3
  FROM test1
  WHERE col1 = 3;
  DBMS_OUTPUT.PUT_LINE
    ('You have extracted the following row of data:');
  DBMS_OUTPUT.PUT_LINE(var1);
  DBMS_OUTPUT.PUT_LINE(var2);
  DBMS_OUTPUT.PUT_LINE(var3);

  SELECT col1, col2, col3 // Second extraction into (same) variables
  INTO var1, var2, var3
  FROM test1
  WHERE col1 = 1;
  DBMS_OUTPUT.PUT_LINE
    ('You have extracted the following row of data:');
  DBMS_OUTPUT.PUT_LINE(var1);
  DBMS_OUTPUT.PUT_LINE(var2);
  DBMS_OUTPUT.PUT_LINE(var3);
END;
/
```

```
You have extracted the following row of data:
3
This is row 3
03-SEP-08
```

```
You have extracted the following row of data:
1
This is row 1
01-SEP-08
```

**Procedure successfully completed.**

### Key Points:

This works because although we are doing two extractions into the same variables we are doing these extractions *at different times* – which is fine because the variables will simply overwrite the previous set of values (from the first extraction). The problems only occur when we get multiple rows of data from the *same extraction process* (query) – then we need cursors (see later code pod).

**Free Sample 12: Using a SQL/XML Query**

Here we introduce a new table...

```
SELECT * FROM Treatment;
```

DOGID	APPOINTMENT	DETAILS	PRICE	VET
1	01-JAN-08	Annual booster jab	25.99	Victor
1	01-JUN-08	Flea Removal	9.99	Vicky
2	31-MAR-08	Annual booster jab	25.99	Valerie
2	01-JUN-08	Eye Drops	15.99	Victor
3	01-JUL-08	General checkup	29.99	Vicky

```
SELECT dogid AS Dog_Identifier,
       XMLELEMENT(NAME Date_Of_Treatment, Appointment),
       XMLELEMENT(NAME Details,
                  XMLELEMENT(NAME Description, Details),
                  XMLELEMENT(NAME Implementation,
                              XMLELEMENT(NAME Price, Price),
                              XMLELEMENT(NAME Vet, Vet)))
       AS XML_Treatment
FROM Treatment;
```

Which produces...

```
DOG_IDENTIFIER
-----
XMLELEMENT(NAMEDATE_OF_TREATMENT, APPOINTMENT)
-----
XML_TREATMENT
-----
          1
<DATE_OF_TREATMENT>01-JAN-08</DATE_OF_TREATMENT>
<DETAILS><DESCRIPTION>Annual booster
jab</DESCRIPTION><IMPLEMENTATION><PRICE>25.99</PRICE>...

          1
<DATE_OF_TREATMENT>01-JUN-08</DATE_OF_TREATMENT>
<DETAILS><DESCRIPTION>Flea
Removal</DESCRIPTION><IMPLEMENTATION><PRICE>9.99</PRICE>...

          2
<DATE_OF_TREATMENT>31-MAR-08</DATE_OF_TREATMENT>
<DETAILS><DESCRIPTION>Annual booster
jab</DESCRIPTION><IMPLEMENTATION><PRICE>25.99</PRICE>...

          2
<DATE_OF_TREATMENT>01-JUN-08</DATE_OF_TREATMENT>
<DETAILS><DESCRIPTION>Eye
Drops</DESCRIPTION><IMPLEMENTATION><PRICE>15.99</PRICE>...

          3
<DATE_OF_TREATMENT>01-JUL-08</DATE_OF_TREATMENT>
<DETAILS><DESCRIPTION>General
checkup</DESCRIPTION><IMPLEMENTATION><PRICE>29.99</PRICE>...
```