

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

Student Workbook

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

Contributing Authors: Danielle Hopkins, Julie Johnson, Rob Roselius, and Robert Seitz

Published by ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, CO 80112.

Editor: Jan Waleri

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Roger Jones, Jim McNally, and Kevin Smith, and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2011 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photo-copying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to ITCourseware, LLC., 7245 South Havana Street, Suite 100, Centennial, Colorado, 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

| | |
|--|----|
| Chapter 1 - Course Introduction | 9 |
| Course Objectives | 10 |
| Course Overview | 12 |
| Using the Workbook | 13 |
| Suggested References | 14 |
| Sample Database | 16 |
| Chapter 2 - Triggers | 21 |
| Beyond Declarative Integrity | 22 |
| Triggers | 24 |
| Types of Triggers | 26 |
| Row-Level Triggers | 28 |
| Trigger Predicates | 30 |
| Trigger Conditions | 32 |
| Using SEQUENCEs | 34 |
| Cascading Triggers and Mutating Tables | 36 |
| Generating an Error | 38 |
| Maintaining Triggers | 40 |
| Labs | 42 |
| Chapter 3 - PL/SQL Variables and Datatypes | 45 |
| Anonymous Blocks | 46 |
| Declaring Variables | 48 |
| Datatypes | 50 |
| Subtypes | 52 |
| Character Data | 54 |
| Dates and Timestamps | 56 |
| Date Intervals | 58 |
| Anchored Types | 60 |
| Assignment and Conversions | 62 |
| Selecting into a Variable | 64 |
| Returning into a Variable | 66 |
| Labs | 68 |

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

| | |
|--|-----|
| Chapter 4 - PL/SQL Syntax and Logic | 71 |
| Conditional Statements — IF/THEN | 72 |
| Conditional Statements — CASE | 74 |
| Comments and Labels | 76 |
| Loops | 78 |
| WHILE and FOR Loops | 80 |
| SQL in PL/SQL | 82 |
| Local Procedures and Functions | 84 |
| Labs | 86 |
| Chapter 5 - Stored Procedures and Functions | 89 |
| Stored Subprograms | 90 |
| Creating a Stored Procedure | 92 |
| Procedure Calls and Parameters | 94 |
| Parameter Modes | 96 |
| Named Parameter Notation | 98 |
| Default Arguments | 100 |
| Creating a Stored Function | 102 |
| Stored Functions and SQL | 104 |
| Invoker's Rights | 106 |
| Labs | 108 |
| Chapter 6 - Exception Handling | 111 |
| SQLCODE and SQLERRM | 112 |
| Exception Handlers | 114 |
| Nesting Blocks | 116 |
| Scope and Name Resolution | 118 |
| Declaring and Raising Named Exceptions | 120 |
| User-Defined Exceptions | 122 |
| Labs | 124 |
| Chapter 7 - Records, Collections, and User-Defined Types | 127 |
| Record Variables | 128 |
| Using the %ROWTYPE Attribute | 130 |
| User-Defined Object Types | 132 |
| VARRAY and Nested TABLE Collections | 134 |
| Using Nested TABLEs | 136 |
| Using VARRAYs | 138 |

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

| | |
|--|-----|
| Collections in Database Tables | 140 |
| Associative Array Collections | 142 |
| Collection Methods | 144 |
| Iterating Through Collections | 146 |
| Labs | 148 |
| Chapter 8 - Cursors | 151 |
| Multi-Row Queries | 152 |
| Declaring and Opening Cursors | 154 |
| Fetching Rows | 156 |
| Closing Cursors | 158 |
| The Cursor FOR Loop | 160 |
| FOR UPDATE Cursors | 162 |
| Cursor Parameters | 164 |
| The Implicit (SQL) Cursor | 166 |
| Labs | 168 |
| Chapter 9 - Bulk Operations | 171 |
| Bulk Binding | 172 |
| BULK COLLECT Clause | 174 |
| FORALL Statement | 176 |
| FORALL Variations | 178 |
| Bulk Returns | 180 |
| Bulk Fetching with Cursors | 182 |
| Labs | 184 |
| Chapter 10 - Using Packages | 187 |
| Packages | 188 |
| Oracle-Supplied Packages | 190 |
| The DBMS_OUTPUT Package | 192 |
| The DBMS_UNUTILITY Package | 194 |
| The UTL_FILE Package | 196 |
| Creating Pipes with DBMS_PIPE | 198 |
| Writing to and Reading from a Pipe | 200 |
| The DBMS_METADATA Package | 202 |
| XML Packages | 204 |
| Networking Packages | 206 |
| Other Supplied Packages | 208 |
| Labs | 210 |

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

| | |
|--|-----|
| Chapter 11 - Creating Packages | 213 |
| Structure of a Package | 214 |
| The Package Interface and Implementation | 216 |
| Package Variables and Package State | 218 |
| Overloading Package Functions and Procedures | 220 |
| Forward Declarations | 222 |
| Strong REF CURSOR Variables | 224 |
| Weak REF CURSOR Variables | 226 |
| Labs | 228 |
| Chapter 12 - Working with LOBs | 231 |
| Large Object Types | 232 |
| Oracle Directories | 234 |
| LOB Locators | 236 |
| Internal LOBs | 238 |
| External LOBs | 240 |
| Temporary LOBs | 242 |
| The DBMS_LOB Package | 244 |
| Labs | 246 |
| Chapter 13 - Maintaining PL/SQL Code | 249 |
| Privileges for Stored Programs | 250 |
| Data Dictionary | 252 |
| PL/SQL Stored Program Compilation | 254 |
| Conditional Compilation | 256 |
| Compile-Time Warnings | 258 |
| The PL/SQL Execution Environment | 260 |
| Dependencies and Validation | 262 |
| Maintaining Stored Programs | 264 |
| Labs | 266 |
| Appendix A - Dynamic SQL | 269 |
| Generating SQL at Runtime | 270 |
| Native Dynamic SQL vs. DBMS_SQL Package | 272 |
| The EXECUTE IMMEDIATE Statement | 274 |
| Using Bind Variables | 276 |
| Multi-row Dynamic Queries | 278 |

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

| | |
|---|-----|
| Bulk Operations with Dynamic SQL | 280 |
| Using DBMS_SQL | 282 |
| DBMS_SQL Subprograms | 284 |
| Appendix B - PL/SQL Versions, Datatypes and Language Limits | 287 |
| Appendix C - Oracle 10g Supplied Packages | 295 |
| Solutions | 307 |
| Index | 339 |

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- ※ Create triggers on database tables.
- ※ Use PL/SQL's datatypes for database and program data.
- ※ Use program structure and control flow to design and write PL/SQL programs.
- ※ Create PL/SQL stored procedures and functions.
- ※ Write robust programs that handle runtime exceptions.
- ※ Use PL/SQL's collection datatypes.
- ※ Use cursors to work with database data.
- ※ Use the packages supplied with Oracle.
- ※ Design and write your own packages.
- ※ Maintain and evolve your PL/SQL programs.
- ※ Manage the security of your stored PL/SQL programs.

COURSE OVERVIEW

- * **Audience:** This course is designed for database application developers programming in an Oracle environment.
- * **Prerequisites:** A working knowledge of the SQL language, as well as a solid understanding of 3GL programming is required.
- * **Classroom Environment:**
 - One workstation per student.
 - Oracle Server and SQL*Plus.

USING THE WORKBOOK

This workbook design is based on a page-pair, consisting of a Topic page and a Support page. When you lay the workbook open flat, the Topic page is on the left and the Support page is on the right. The Topic page contains the points to be discussed in class. The Support page has code examples, diagrams, screen shots and additional information. **Hands On** sections provide opportunities for practical application of key concepts. **Try It** and **Investigate** sections help direct individual discovery.

In addition, there is an index for quick look-up. Printed lab solutions are in the back of the book as well as on-line if you need a little help.

The Topic page provides the main topics for classroom discussion.

The Support page has additional information, examples and suggestions.

JAVA SERVLETS

THE SERVLET LIFE CYCLE

- * The servlet container controls the life cycle of the servlet.
- When the first request is received, the container loads the servlet class.
- The container uses a separate thread to call the `init()` method.
- The container calls the `destroy()` method.

Topics are organized into first (*), second (>) and third (■) level points.

- As with Java's `finalize()` method, don't count on this being called.
- * Override one of the `init()` methods for one-time initializations, instead of using a constructor.
- The simplest form takes no parameters.

```
public void init() { ... }
```

- If you need to know container-specific configuration information, use the other version.

```
public void init(ServletConfig config) { ... }
```

- Whenever you use the `ServletConfig` approach, always call the superclass method, which performs additional initializations.

```
super.init(config);
```

CHAPTER 2

SERVLET BASICS

The Support page has additional information, examples and suggestions.

Hands On:

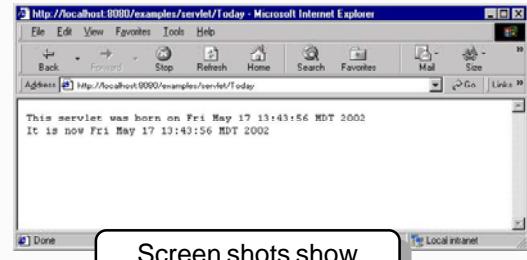
Add an `init()` method to your `Today` servlet that initializes along with the current date:

```
Today.java
...
public class Today extends GenericServlet {
    private Date bornOn;
    public void service(ServletRequest request,
                        ServletResponse response) throws ServletException, IOException
    {
        ...
        today = new Date();
        out.println("This servlet was born on " + bornOn.toString());
        out.println("It is now " + today.toString());
    }
}
```

Code examples are in a fixed font and shaded. The on-line file name is listed above the shaded area.

Callout boxes point out important parts of the example code.

The `init()` method is called when the servlet is loaded into the container.



Screen shots show examples of what you should see in class.

Pages are numbered sequentially throughout the book, making lookup easy.

SUGGESTED REFERENCES

Allen, Christopher. 2004. *Oracle Database 10g PL/SQL 101*. McGraw-Hill Osborne, Emeryville, CA.
ISBN 0072255404

Boardman, Susan, Melanie Caffrey, Solomon Morse, and Benjamin Rosenzweig. 2002. *Oracle Web Application Programming for PL/SQL Developers*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0130477311

Date, C.J and Hugh Darwen. 1996. *A Guide to SQL Standard*. Addison-Wesley, Boston, MA.
ISBN 0201964260

Date, C.J. 2003. *An Introduction to Database Systems*. Addison-Wesley, Boston, MA.
ISBN 0321197844

Feuerstein, Steven, Charles Dye, and John Beresniewicz. 1998. *Oracle Built-in Packages*. O'Reilly and Associates, Sebastopol, CA. ISBN 1565923758

Feuerstein, Steven. 2007. *Oracle PL/SQL Best Practices, Second Edition*. O'Reilly and Associates, Sebastopol, CA. ISBN 0596514107

Feuerstein, Steven. 2000. *Oracle PL/SQL Developer's Workbook*. O'Reilly and Associates, Sebastopol, CA. ISBN 1565926749

Feuerstein, Steven and Bill Pribyl. 2005. *Oracle PL/SQL Programming, Fourth Edition*. O'Reilly and Associates, Sebastopol, CA. ISBN 0596009771

Loney, Kevin. 2004. *Oracle Database 10g: The Complete Reference*. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072253517

McDonald, Connor, Chaim Katz, Christopher Beck, Joel R. Kallman, and David C. Knox. 2004. *Mastering Oracle PL/SQL: Practical Solutions*. Apress, Berkeley, CA.
ISBN 1590592174

Pribyl, Bill. 2001. *Learning Oracle PL/SQL*. O'Reilly and Associates, Sebastopol, CA.
ISBN 0596001800

Price, Jason. 2004. *Oracle Database 10g SQL*. McGraw-Hill Osborne, Emeryville, CA.
ISBN 0072229810

Rosenzweig, Benjamin and Elena Silvestrova Rakhimov. 2008. *Oracle PL/SQL by Example*. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0137144229

Urman, Scott and Michael McLaughlin. 2004. *Oracle Database 10g PL/SQL Programming*. McGraw-Hill Osborne, Emeryville, CA. ISBN 0072230665

*www.oracle.com/technology/tech/pl_sql
tahiti.oracle.com*

SAMPLE DATABASE

Our company is a hardware/software retailer with stores in several cities.

We keep track of each person's name, address, and phone. In addition, if a person is an employee, we must record in which store he or she works, the supervisor's ID, the employee's title, pay amount, and compensation type ("Hourly," "Salaried," etc.)

Sometimes a customer will fill out an order, which requires an invoice number. Each invoice lists the store and the customer's ID. We record the quantity of each item on the invoice and any discount for that item. We also keep track of how much the customer has paid on the order.

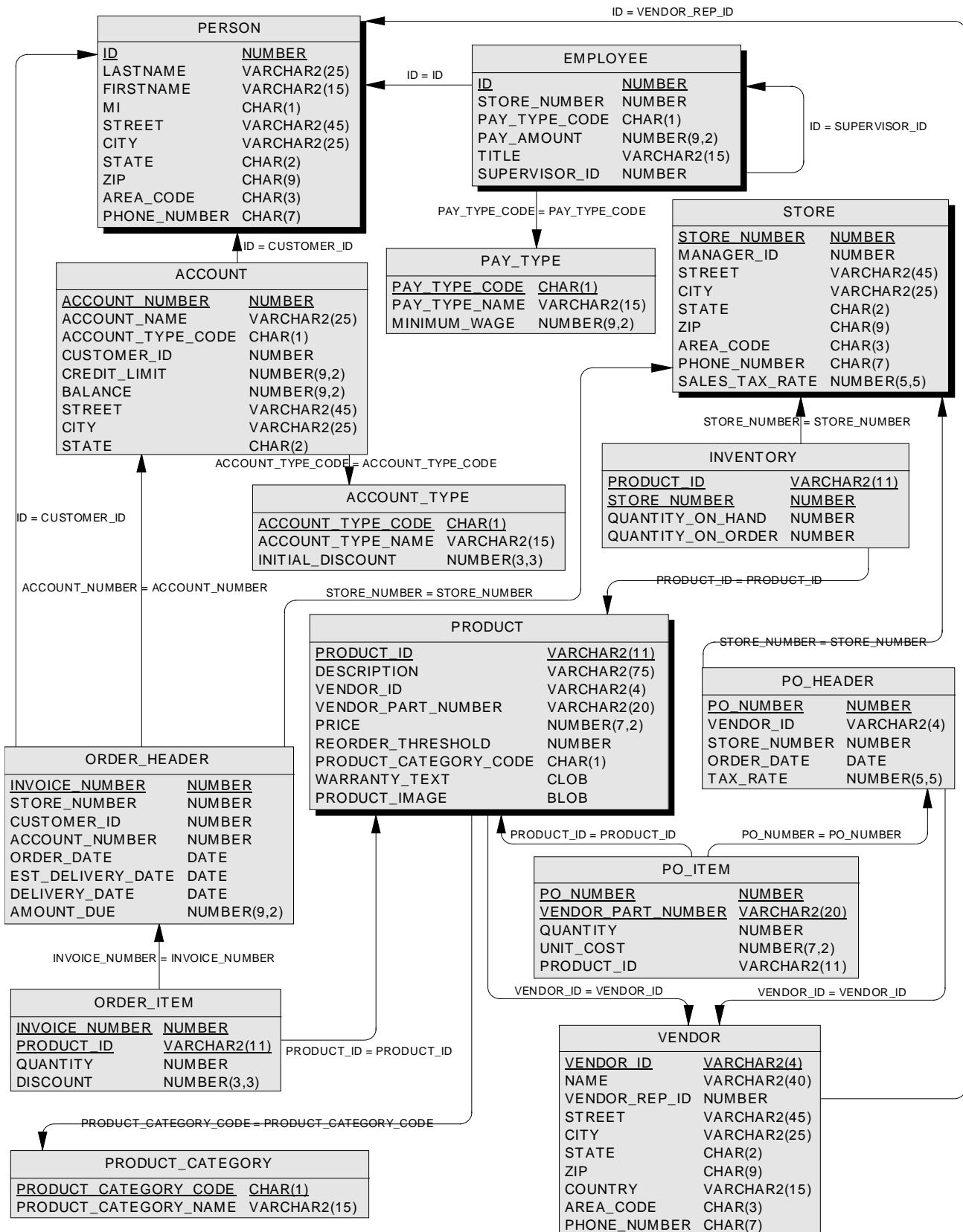
When a credit account is used for an order, the balance for the account must be updated to reflect the total amount of the invoice (including tax). The salesperson verifies (with a phone call) that the person is a valid user of the account.

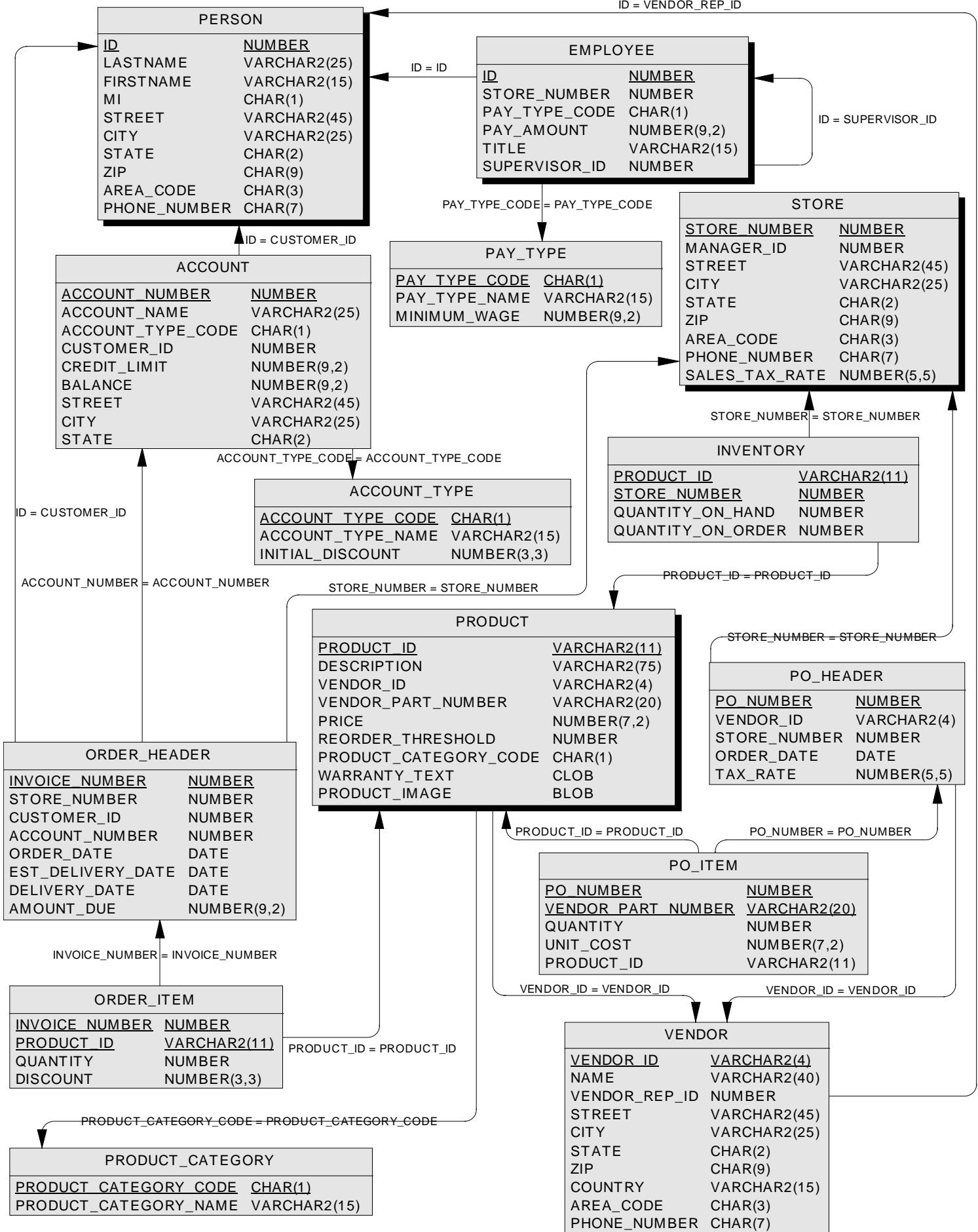
Each product we sell has a product ID and description. In addition, we keep track of the vendor we purchase that product from, the ID for that product, and the category ("Software," "Peripheral," or "Service"). We also store the address, phone, and sales rep ID for each individual vendor.

We keep track of how many items are on hand at each store and how many each store has on order. When an item is sold, the inventory is updated.

We maintain the address, phone number, and manager ID for each store. Each store has a unique store number. We record the sales tax rate for that store.

When a store runs low on a product, we create a purchase order. Each purchase order in our company has a unique PO number. A PO is sent to a single vendor, from which we might be ordering several items, each at a specific unit cost. The inventory reflects the sale or order of any item in a store.





INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

CHAPTER 3 - PL/SQL VARIABLES AND DATATYPES

OBJECTIVES

- * Create anonymous blocks for testing and scripting.
- * Choose appropriate datatypes and construct correct declarations for PL/SQL program variables.
- * Convert between basic datatypes.
- * Retrieve a database column value into a PL/SQL program variable.
- * Capture data from a DML statement with the **RETURNING** clause.

ANONYMOUS BLOCKS

- ※ PL/SQL code can be stored and run from the database in triggers, stored procedures, stored functions, and within packages.
- ※ PL/SQL code can also be run from an anonymous block.
- ※ An anonymous block is syntax checked and executed each time it is issued.
 - The term *SQL script* typically refers to a file that contains SQL statements as well as PL/SQL anonymous blocks.
 - These scripts may be used to help administer the database, or for work during development.
 - These scripts are not part of the database structure and cannot be used from application code.
- ※ An anonymous block has an optional declaration section followed by a code section.

```
[DECLARE  
    variable declarations]  
BEGIN  
    statements  
END;
```

- You may omit the declaration section if you are not declaring any local variables.
- A block must contain at least one statement, even if it does nothing, such as **NULL;**.
- ※ Anonymous blocks are often used to test other PL/SQL stored programs.

The following script contains SQL statements as well as an anonymous block. This script can be modified and executed again, from the file on the computer's hard drive.

```
end_of_month.sql
SELECT credit_limit, balance
  FROM account
 WHERE account_number = 1660
/

DECLARE
    limit NUMBER;
    bal NUMBER;
    interest NUMBER;
BEGIN
    SELECT credit_limit, balance
      INTO limit, bal
      FROM account
     WHERE account_number = 1660;

    IF limit < bal * 2 THEN
        interest := .2;
    ELSE
        interest := .1;
    END IF;

    UPDATE account
       SET balance = balance * (1 + interest)
     WHERE account_number = 1660;
END;
/

SELECT credit_limit, balance
  FROM account
 WHERE account_number = 1660
/
```

DECLARING VARIABLES

- * You must declare all PL/SQL variables in a declaration section at the beginning of the program.
- * The syntax of a PL/SQL variable declaration closely resembles a column specification of a SQL DDL statement:

```
identifier [CONSTANT] datatype [DEFAULT value] [NOT NULL ] ;
```

- You can declare only one variable per statement.
- * You can initialize a variable at declaration:

```
pay_type CHAR(1) := 'S';
min_wage NUMBER DEFAULT 5.15;
```
- Use the **DEFAULT** keyword for values that typically don't change.
- * You must initialize constants when declaring them.

```
PI CONSTANT NUMBER := 3.14159;
```

- You can use any expression to initialize the constant.
- Constant variables are read-only.
- * **NOT NULL** prevents you from assigning **NULL** to a variable, throwing a **VALUE_ERROR** exception if you try.

```
salary NUMBER NOT NULL := 0;
```

You can create identifiers up to 30 characters in length. They must begin with a letter, followed by any other letters, numbers, \$, #, or _. Trailing or contiguous \$, #, or _ are not allowed. Never use a PL/SQL reserved word as a variable name.

Identifiers

```
first_name      -- legal
pay$amount      -- legal
emp#count       -- legal
emp:sal         -- illegal (: not allowed)
fname_           -- illegal (trailing _)
pay$$amount     -- illegal (contiguous $)
raise           -- illegal (reserved word)
```

DATATYPES

- * PL/SQL provides four categories of datatypes for declaring variables:
 - *Scalar* — Single-valued, fundamental types.
 - Datatypes corresponding to database column types.
 - Datatypes unique to PL/SQL.
 - *Composite* — Collections of one or more scalar types.
 - **RECORD** — A group of arbitrary scalar types.
 - **TABLE** — An unbounded, noncompact list of values of a specific type.
 - **VARRAY** — A bounded, compact list of values of a specific type.
 - *Reference* — A reference to a cursor or object type.
 - *Locator* — The location of a LOB (Large OBject).
- * Some PL/SQL datatypes aren't available in Oracle database tables, such as **BOOLEAN** and **BINARY_INTEGER**.
 - Size limits differ between some PL/SQL types and their corresponding Oracle internal types.
- * Constrained types, such as **NUMBER(*p, s*)**, **CHAR(*n*)**, and **VARCHAR2(*n*)**, will raise a **VALUE_ERROR** exception if you try to assign a value that is too big.

```
zip_code CHAR(5) := '86751-4932'; --will raise VALUE_ERROR
```

| Type | PL/SQL | Subtypes | Constraint | Oracle Database |
|-----------------------|---------------------------------|---|--|---------------------------------|
| NUMBER(<i>p,s</i>) | 1E-129..1E+125 | DEC DECIMAL DOUBLE PRECISION FLOAT INTEGER INT NUMERIC REAL SMALLINT | | 1E-129..1E+125 |
| CHAR(<i>n</i>) | 1..32767 Bytes | CHARACTER | | 1..2000 Bytes |
| VARCHAR2(<i>n</i>) | 1..32767 Bytes | VARCHAR STRING | | 1..4000 Bytes |
| NCHAR(<i>n</i>) | 1..32767 Bytes | | | 1..2000 Bytes |
| NVARCHAR2(<i>n</i>) | 1..32767 Bytes | | | 1..4000 Bytes |
| DATE | 1/1/4712 BC 12/31/9999 AD | | | 1/1/4712 BC 12/31/9999 AD |
| UROWID | | | | |
| MLSLABEL | | | | |
| BLOB | <i>Locator</i> | | | 1..4294967295 Bytes |
| CLOB | <i>Locator</i> | | | 1..4294967295 Bytes |
| NCLOB | <i>Locator</i> | | | 1..4294967295 Bytes |
| BFILE | <i>Locator</i> | | Read-Only | 1..4294967295 Bytes |
| LONG | 1..32760 Bytes | | | 1..2147483647 Bytes |
| RAW | 1..32767 Bytes | | | 1..2000 Bytes |
| LONG RAW | 1..32760 Bytes | | | 1..2147483647 Bytes |
| BOOLEAN | TRUE, FALSE, NULL | | | not supported |
| BINARY_INTEGER | -2147483648.. +2147483647 | NATURAL POSITIVE NATURALN (non NULL) POSITIVEN (non NULL) SIGNTYPE | 0..2147483647 1..2147483647 0..2147483647 1..2147483647 -1, 0, 1 | not supported |
| PLS_INTEGER | -2147483648.. +2147483647 | | | not supported |
| BINARY_FLOAT | 32-bit floating point number | BINARY_FLOAT_NAN BINARY_FLOAT_INFINITY BINARY_FLOAT_MAX_NORMAL BINARY_FLOAT_MIN_NORMAL BINARY_FLOAT_MAX_SUBNORMAL BINARY_FLOAT_MIN_SUBNORMAL | | 32-bit floating point number |
| BINARY_DOUBLE | 64-bit floating point number | BINARY_DOUBLE_NAN BINARY_DOUBLE_INFINITY BINARY_DOUBLE_MAX_NORMAL BINARY_DOUBLE_MIN_NORMAL BINARY_DOUBLE_MAX_SUBNORMAL BINARY_DOUBLE_MIN_SUBNORMAL | | 63-bit floating point number |

SUBTYPES

- * A **SUBTYPE** is a specialized name, based on a scalar or user-defined datatype.

```
SUBTYPE type_name IS base_type [(constraint)] [NOT NULL];
```

- You can define additional precision and scale constraints for numbers, or maximum size for characters.

```
SUBTYPE bday_type IS DATE NOT NULL;
SUBTYPE delivery_date_type IS bday_type;
SUBTYPE ssn_type IS CHAR(9) NOT NULL;
SUBTYPE currency_type IS NUMBER(9,2);
```

- * Oracle predefines several subtypes:

- **DECIMAL**
- **INTEGER**
- **REAL**
- **NATURAL**
- **SIGNTYPE**
- etc...

- * After you define a **SUBTYPE**, you can declare a variable of that type:

```
SUBTYPE ssn_type IS CHAR(9) NOT NULL;
person_ssn ssn_type := '000000000';
```

Different subtypes are convertible if they have the same base type or are in the same datatype family, and if assignment doesn't result in data truncation:

subtypes.sql

```
DECLARE
    SUBTYPE sentence_type IS VARCHAR2(1000);
    SUBTYPE word_type IS CHAR(10);
    description sentence_type;
    word word_type;
BEGIN
    description := 'Flower Pot';    --10 bytes long
    word := description;           --description and word are in same
                                    --datatype family.
    description := 'Baseball Glove';
    word := description;          --description now exceeds 10 bytes, so
                                    --Oracle throws a VALUE_ERROR exception.
END;
/
```

CHARACTER DATA

- * **CHAR** and **VARCHAR2** define character data as a sequence of characters enclosed inside single quotes.

```
fname VARCHAR2(15) := 'Robert';
```

- * If character data contains a literal single quote, you may escape it with another single quote:

```
msg VARCHAR2(45) := 'What ''s up, doc?';
```

- * Starting with Oracle 10g, you can use the quote operator with your own delimiter.

```
q'c text inbetween c'
```

- This lets you avoid escaping many single quotes, improving readability.

```
msg := q'!Employee #7265 is O'Connor!';
```

- Choose any delimiter that is not present in the character data.
- If the opening delimiter is (, [, {, <, then close it with),], }, and >, respectively.

```
msg := q'{Employee #7265 is O'Connor}';
```

- * **CHARs** are space-padded if needed; **VARCHAR2s** are not.
- When comparing two **CHAR** values, the shorter value will be padded to the length of the larger value prior to comparison.

```
vars.sql
```

```
DECLARE
    lname VARCHAR2(25);
    fname VARCHAR2(10);
    city VARCHAR2(10) := 'Denver';
    city2 CHAR(10) := 'Denver';
    result BOOLEAN;
BEGIN
    lname := 'Smith ' -- String holds 7 bytes (note the extra spaces)
    --fname := 'Christopher'; -- Too long for this datatype
    IF city = city2 THEN -- FALSE (city2 is space-padded, city isn't)
        result := TRUE;
    ELSE
        result := FALSE;
    END IF;

    IF city2 = 'Denver'      ' THEN -- TRUE (Comparing a CHAR to a CHAR,
                                -- the smaller CHAR gets space padded)
        result := TRUE;
    ELSE
        result := FALSE;
    END IF;

    IF RTRIM(city2) = 'Denver' THEN -- TRUE (remove space padding first.)
        result := TRUE;
    ELSE
        result := FALSE;
    END IF;
END;
/
```

NCHAR(size) and **NVARCHAR2(size)** support Unicode characters. The DBA defines the default national character set upon database creation. Oracle implicitly converts **CHAR** and **NCHAR**, as well as **VARCHAR2** and **NVARCHAR2** in PL/SQL statements and expressions, but could cause truncation based on the data in the nationalized strings.

Use the **nq** quoting mechanism with **NCHAR** and **NVARCHAR2**:

```
msg NVARCHAR2(200) := nq'! Employee #9999 is BNolé!';
```

DATES AND TIMESTAMPS

- * The **DATE** datatype stores the century, year, month, day, hour, minute, and second of a date.
 - Two dates are equal if they contain the same information down to the second.
 - Use the **trunc(date, format_specifier)** function to compare two dates to a certain precision:

```
IF trunc(order_date, 'dd') = trunc(sysdate-10, 'dd')
    --if order was made sometime 10 days ago
IF trunc(order_date, 'yyyy') = trunc(sysdate, 'yyyy')
    --if order was made this year
```

- * Oracle 9*i* introduced the **TIMESTAMP(precision)** datatype as an extension of **DATE**, additionally storing fractional seconds.
 - Valid *precision* values range from **0** to **9**; default **6**.
- * **TIMESTAMP(precision) WITH TIME ZONE** includes the time displacement compared with Coordinated Universal Time (UTC).
 - Two **TIMESTAMP WITH TIME ZONE** values are equal if they represent the same UTC time.
- * **TIMESTAMP(precision) WITH LOCAL TIME ZONE** is the same, except...
 - When inserting the date into a table, Oracle automatically alters the date from the session's time zone to the database's time zone.
 - Displacement information is not stored.

The **NLS_DATE_FORMAT**, **NLS_TIMESTAMP_FORMAT**, and **NLS_TIMESTAMP_TZ_FORMAT** parameters define the default display format for dates, timestamps, and timestamps with time zones, respectively.

```
ALTER SESSION SET nls_date_format = 'mm-dd-yyyy'
```

You can also specify the time zone with a name or abbreviation, which are defined in the **V\$TIMEZONE_NAMES** Data Dictionary view.

```
shipping_date TIMESTAMP WITH TIME ZONE := '2003-01-12 3:00:00 US/Hawaii';
```

Try it:

Create a table with a **TIMESTAMP WITH TIME ZONE** column, as well as the **TIMESTAMP WITH LOCAL TIME ZONE** column.

```
CREATE TABLE times(
    x TIMESTAMP WITH TIME ZONE,
    y TIMESTAMP WITH LOCAL TIME ZONE
);
```

Populate a record with the **sysdate** for both columns:

```
INSERT INTO times
VALUES (sysdate, sysdate);
```

Now query the table:

```
SELECT * FROM times;
```

Alter your session's time zone, then query the table again. Did you notice the time change for **TIMESTAMP WITH LOCAL TIME ZONE**?

```
ALTER SESSION SET time_zone='CET'; --Poland time zone
```

```
SELECT * FROM times;
```

DATE INTERVALS

- ※ Oracle 9*i* introduced date intervals to ease the task of manipulating periods of time.
- ※ **INTERVAL YEAR(*precision*) TO MONTH** stores year and month intervals.
 - *precision* is the number of digits for the year field, 0-9; default 2.
 - '*y-m*' is the default pattern to provide a default value.
- ※ You can assign a year, month, or combination value:

```
age INTERVAL YEAR(9) TO MONTH := '5-3';      --5 years, and 3
                                         --months
```

- ※ **INTERVAL DAY(*day_precision*) TO SECOND(*second_precision*)** stores day, hour, minute, and second intervals.
 - The precision values for both days and seconds range from 0-9, with defaults of 2 and 6, respectively.
 - The default format is '*days hours:minutes:seconds*'.

```
ship_time INTERVAL DAY(3) TO SECOND := '200 10:5:2';
                                         --200 days, 10 hours, 5 minutes, 2 seconds.
```

- ※ Use an interval expression to create an **INTERVAL** value:

```
age := INTERVAL '10' YEAR;                  --10 years
age := INTERVAL '2-3' YEAR TO MONTH;    --2 years, 3 months
ship_time := INTERVAL '15' HOUR;          --15 hours
ship_mate := INTERVAL '50' MINUTE;        --50 minutes
```

You may perform arithmetic with dates and intervals:

DATE + INTERVAL = DATE
INTERVAL + INTERVAL = INTERVAL
DATE - INTERVAL = DATE
DATE - DATE = number
INTERVAL - INTERVAL = INTERVAL
INTERVAL * numeric = INTERVAL
numeric * INTERVAL = INTERVAL
INTERVAL / numeric = INTERVAL
DATE + numeric = DATE
DATE - numeric = DATE

When subtracting a date from another date to find the interval between them, the result will actually be an ordinary numeric value (a number of days, including fractional days), not an **INTERVAL**. If you need to store the result as an **INTERVAL** variable or column value, use one of two conversion functions:

NUMTODSINTERVAL(*numeric, unit*) converts the number to an **INTERVAL DAY TO SECOND** value.

NUMTOYMINTERVAL(*numeric, unit*) converts the number to an **INTERVAL YEAR TO MONTH** value.

For each, *unit* specifies which unit *numeric* represents. For **NUMTODSINTERVAL()**, it must be one of '**DAY**', '**HOUR**', '**MINUTE**', or '**SECOND**'. For **NUMTOYMINTERVAL()**, it must be either '**YEAR**' or '**MONTH**'.

```
DECLARE
    daycount NUMBER;
    date1 DATE := SYSDATE;
    date2 DATE := '25-DEC-2005';
    daysuntil INTERVAL DAY TO SECOND;
    yearsuntil INTERVAL YEAR TO MONTH;
BEGIN
    daycount := date2 - date1;
    daysuntil := NUMTODSINTERVAL( daycount, 'DAY' );
    yearsuntil := NUMTOYMINTERVAL( daycount/365, 'YEAR' );
    ...

```

ANCHORED TYPES

- * Variables often store data read in from the database.
 - You should declare these variables with the same datatype and size as the column definition in the Data Dictionary:

```
curr_partno VARCHAR2(20);
```

- * If you change a table's definition, then you must change all dependent PL/SQL code.
 - For example, if you alter **product** table's **vendor_part_number** size from 20 to 25, all dependent code also has to change.

```
curr_partno VARCHAR2(25);
```

- - You must edit all declarations of this form manually.

- * Using the **%TYPE** attribute of a column in your declaration removes the need for manual modification later.

```
curr_partno product.vendor_part_number%TYPE;
```

- * The compiler automatically determines the datatype for the variable.
 - Dependent code automatically reflects changes made to a column's definition, without manual editing.
- * You can also use **%TYPE** with any previously defined variable:

```
new_partno curr_partno%TYPE;
```

You must know the name, datatype, and size of columns when you declare PL/SQL program variables for use in DML statements:

```
vendor1.sql
DECLARE
    curr_partno  VARCHAR2(20);
    new_partno   VARCHAR2(20);
    vend_id      VARCHAR2(4);
BEGIN
    SELECT vendor_id, vendor_part_number
        INTO vend_id, curr_partno
        FROM product
       WHERE product_id = 'BROD0000020';

    new_partno := vend_id || '-' || curr_partno;

    UPDATE product
        SET vendor_part_number = new_partno
       WHERE product_id = 'BROD0000020';
END;
/
```

With anchored types, you only need to know the column names:

```
vendor2.sql
DECLARE
    curr_partno  product.vendor_part_number%TYPE;
    new_partno   curr_partno%TYPE;
    vend_id      product.vendor_id%TYPE;
BEGIN
    SELECT vendor_id, vendor_part_number
        INTO vend_id, curr_partno
        FROM product
       WHERE product_id = 'BROD0000020';

    new_partno := vend_id || '-' || curr_partno;

    UPDATE product
        SET vendor_part_number = new_partno
       WHERE product_id = 'BROD0000020';
END;
/
```

ASSIGNMENT AND CONVERSIONS

- * The PL/SQL operator := assigns a value to a variable.

```
variable := expression;
```

- * The expression on the right-hand side of the assignment must yield a value whose datatype is the same as, or is compatible with, the datatype of the variable on the left-hand side.
- * Just as in SQL, some types are implicitly convertible to others.

```
order_date := '18-JUN-00';
```

```
lname := lname || ' ' || fredcount;
```

- Implicit conversion uses default input and output formats.
- * Oracle provides functions for explicitly converting from one datatype to another.

```
order_date := TO_DATE('06/18/00', 'MM/DD/YY');
```

```
lname := lname || ' ' || TO_CHAR(fredcount, 'RN');
```

```
place_order.sql
DECLARE
    xmas_day  order_header.order_date%TYPE;
    next_invoice  order_header.invoice_number%TYPE;
BEGIN
    xmas_day := TO_DATE('December 25, 2006', 'Month DD, YYYY');

    SELECT MAX(invoice_number)
        INTO next_invoice
        FROM order_header;

    next_invoice := next_invoice + 1;

    INSERT INTO order_header (invoice_number, store_number,
        customer_id, order_date, amount_due)
    VALUES (next_invoice, 7, 7881, xmas_day, 0);

    INSERT INTO order_item (invoice_number, product_id,
        quantity, discount)
    VALUES (next_invoice, 'CHYE0000015', 10, 0.1);
END;
/
```

SELECTING INTO A VARIABLE

- * A **SELECT...INTO** statement is another way of assigning a value to a PL/SQL program variable.
- * When you **SELECT** data from the database in PL/SQL, you must provide a variable for each item selected.
 - List the variables in the **INTO** clause, immediately following the **SELECT** clause.

```
SELECT expression1[, expression2...]  
      INTO variable1[, variable2...]  
      FROM table(s)  
      WHERE condition(s);
```

- Oracle will perform implicit conversion of database data to the variables' types, when possible.
- * You must ensure that you retrieve exactly one row.
 - If the query returns no rows, a **NO_DATA_FOUND** error occurs.
 - If the query returns more than one row, a **TOO_MANY_ROWS** error occurs.
 - In either case, Oracle throws a PL/SQL exception.

```
place_order.sql
DECLARE
    xmas_day  order_header.order_date%TYPE;
    next_invoice  order_header.invoice_number%TYPE;
BEGIN
    xmas_day := TO_DATE('December 25, 2006', 'Month DD, YYYY');

    SELECT MAX(invoice_number)
      INTO next_invoice
     FROM order_header;

    next_invoice := next_invoice + 1;

    INSERT INTO order_header (invoice_number, store_number,
        customer_id, order_date, amount_due)
    VALUES (next_invoice, 7, 7881, xmas_day, 0);

    INSERT INTO order_item (invoice_number, product_id,
        quantity, discount)
    VALUES (next_invoice, 'CHYE0000015', 10, 0.1);
END;
/
```

RETURNING INTO A VARIABLE

- * You can retrieve the values of rows affected by an **INSERT**, **UPDATE**, or **DELETE** statement using the **RETURNING...INTO** clause.

```
... RETURN[ING] expression INTO variable[, variable] ...
```

- The expression is typically a list of columns.

```
UPDATE employee
      SET pay_amount = 52000
    WHERE id = 7881
RETURNING pay_amount INTO old_pay;
```

- If the DML statement does not affect any rows, the value of the variables is undefined.
 - Initializing the variables to **NULL** makes it easier to test the results.

```
IF old_pay != NULL AND old_pay < (52000 * .8) THEN
```

```
return.sql
DECLARE
    old_pay employee.pay_amount%TYPE := NULL;
BEGIN
    UPDATE employee
        SET pay_amount = 52000
        WHERE id = 7881
    RETURNING pay_amount INTO old_pay;

    IF old_pay != NULL AND old_pay < (52000 * .8) THEN
        COMMIT;
    ELSE
        ROLLBACK;
    END IF;

END;
/
```

LABS

- 1** Write a program that uses an anchored variable for an employee's title. What happens when you try to set it to 'Sales Representative'? What can you do to solve this problem?
(Solution: *anchored.sql*)

- 2** Write a program that creates variables to hold information about a manager. Use appropriate datatypes to keep track of the manager's first and last names, pay rate, and store number. Assign values to each of these within the body of the program.
(Solution: *manager1.sql*)

- 3** Modify the program from **2** to retrieve values into these variables for person 7881, who manages one of the stores.
(Solution: *manager2.sql*)

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

CHAPTER 7 - RECORDS, COLLECTIONS, AND USER-DEFINED TYPES

OBJECTIVES

- ※ Declare and use **RECORD** variables.
- ※ Create user-defined object types in the database.
- ※ Declare and use nested **TABLEs**, **VARRAYs**, and associative array collections.
- ※ Add collections as columns in a database table.
- ※ Use collection methods to manipulate nested **TABLEs**, **VARRAYs**, and associative array collections.

RECORD VARIABLES

- ※ A **RECORD** groups several fields together in a single variable.
 - Each field in the record may be of any datatype.
- ※ Creating a **RECORD** variable takes two steps:
 1. Define a record type in the **DECLARE** section:

```
TYPE rec_type IS RECORD (
    field1 DATATYPE [NOT NULL] := default
    [, field2 DATATYPE [NOT NULL] := default]
    ...
);
```

2. Declare variables of that type:

```
var_1 rec_type;
var_2 rec_type;
```

- ※ There are three ways to assign values to **RECORD** variables.
 - You can assign each individual field using dot notation.
 - You can assign a **RECORD** variable to another of the same type.
 - You can **SELECT** or **FETCH** data into a record variable.

```
SELECT col1 [, col2 ...] INTO var_1 FROM tab ...
```

This record type groups some selected fields of employee information:

```
TYPE employee_rec IS RECORD (
    empid NUMBER,
    pay_type CHAR(1),
    pay_amount NUMBER(9,2),
    title VARCHAR2(15)
);
```

You can now create different employee variables:

```
emp1 employee_rec;
emp2 employee_rec;
```

To access the different fields within the record, use dot notation:

```
emp1.empid := 6600;
emp1.title := 'Asst Manager';
```

You can select into fields of a record, just as you can with ordinary scalar variables.

```
SELECT id, pay_type_code, pay_amount, title
  INTO emp2.empid, emp2.pay_type, emp2.pay_amount, emp2.title
  FROM employee
 WHERE id = 7111;
```

If the items in the select list match the number and types of the fields in the record, you can select into the record:

```
SELECT id, pay_type_code, pay_amount, title
  INTO emp2
  FROM employee
 WHERE id = 7111;
```

You can assign an entire record to another record variable only if they are declared using the same type name:

```
emp1 := emp2;
```

USING THE %ROWTYPE ATTRIBUTE

- * Use the **%ROWTYPE** attribute of a database table to declare a record whose fields match the columns of the table.

```
empa employee%ROWTYPE;
```

- The variable's fields will have the same names and datatypes as the columns in the database table.

```
empa.pay_type_code := 'T';
```

- * **%ROWTYPE** can also declare a variable anchored to another previously declared **RECORD** or **%ROWTYPE** variable.

```
empb emp%ROWTYPE;
```

- * You can assign from a **%ROWTYPE** to a **RECORD** (and vice versa).
 - The number, order, and datatypes of the **RECORD** variable's fields must match the columns of the database table.

A **%ROWTYPE** variable holds a record from a database table:

```
DECLARE
    empa employee%ROWTYPE;
    empb empa%ROWTYPE;
BEGIN
    SELECT *
        INTO empa
        FROM employee
        WHERE id = 7111;
    ...

```

Again, use dot notation to access individual fields of the record. You can copy an entire record to another compatible variable with direct assignment:

```
empa.pay_amount := empa.pay_amount * 1.05;
empb := empa;
```

USER-DEFINED OBJECT TYPES

- * In addition to types that can be defined in a PL/SQL block of code, Oracle also allows you to create types that are permanently stored in the database as part of your schema.
 - Types stored in the database can be used from PL/SQL code, as well as in the definition of columns in Oracle tables.
- * Object types allow you to group attributes into a datatype, in much the same way that records contain a set of fields.
 - Additionally, objects can contain methods (functions and procedures) that operate on the attributes of an object.
 - These can be written in PL/SQL, or externally in Java or C.
 - Objects can also inherit functionality from other object types, therefore implementing a hierarchy.
- * To use an object type in PL/SQL, it must first be created in the database.

engine_typ.sql

```
CREATE TYPE engine_typ AS OBJECT (
    cylinders NUMBER,
    manufacturer VARCHAR2(30)
);
```

- * You can then use the type in another type declaration, or in a PL/SQL block.

car_typ.sql

```
CREATE TYPE car_typ AS OBJECT (
    make VARCHAR2(30),
    model VARCHAR2(30),
    year CHAR(4),
    engine engine_typ
);
```

- * Objects allow you to model real-world entities and allow you to pass complex data structures to and from stored programs.

A table can be created that contains a user-defined object type as one of its columns. Behind the scenes, Oracle creates all of the necessary tables and keys to maintain the relationships properly.

vehicle.sql

```
CREATE TABLE vehicle (
    vehicle_id CHAR(17),
    date_of_service DATE,
    characteristics car_typ
);
```

A stored procedure can take an object as one of its parameters.

pop_vehicle.sql

```
CREATE OR REPLACE PROCEDURE populate_vehicle (
    vin vehicle.vehicle_id%TYPE,
    in_service vehicle.date_of_service%TYPE,
    car_info car_typ
)
IS
BEGIN
    INSERT INTO vehicle
        VALUES (vin, in_service, car_info);
END;
```

Another PL/SQL block can create an object, and pass it to a procedure.

add_vehicle.sql

```
DECLARE
    my_car car_typ;
BEGIN
    my_car := car_typ('Ford', 'Explorer', '2008', engine_typ(6, 'Ford'));
    populate_vehicle('1FMZU73E08ZA69101', '01-SEP-2008', my_car);
END;
/
```

You can query entire rows that contain objects, or individual pieces of information.

query_vehicle.sql

```
SELECT * FROM vehicle
/
SELECT v.characteristics.engine.cylinders
    FROM vehicle v
/
```

You must include the table name or alias name in the **SELECT** list.

VARRAY AND NESTED TABLE COLLECTIONS

- ※ **VARRAYs** and nested **TABLEs** are collections that store values in a numerical order.
 - **VARRAYs** are similar to arrays in other languages and nested tables are similar to bags or sets.
- ※ A *collection* variable holds an ordered list of values of some type, similar to the one-dimensional arrays found in other languages.
- ※ Values of the elements are accessed using a numeric index.
- ※ You can store both **VARRAYs** and nested **TABLEs** in Oracle tables.
- ※ Using either kind of collection requires several steps:

1. Define the collection type.

```
TYPE salary_t IS TABLE OF NUMBER(9,2);
```

2. Declare a collection variable.

```
empsal salary_t;
```

3. Call the collection *constructor* to create the collection.

```
empsal := salary_t();      -- Create the TABLE.  
                           -- No elements exist yet.
```

4. Create the collection elements.

```
empsal.EXTEND(120);    -- Create 120 elements
```

5. Assign and reference the collection elements.

```
empsal(1) := 8.50;  
empsal(90) := empsal(1);
```

```
collect.sql
```

```
DECLARE
    TYPE salary_t IS TABLE OF NUMBER(9,2);
    emp_sal salary_t;
    emp_sum NUMBER(9,2) := 0;
BEGIN
    emp_sal := salary_t();
    emp_sal.EXTEND(5);

    emp_sal(1) := 5.5;
    emp_sal(2) := 6.5;
    emp_sal(3) := 6;
    emp_sal(4) := 5.5;
    emp_sal(5) := 8.5;

    FOR x IN 1 .. 5 LOOP
        emp_sum := emp_sum + emp_sal(x);
    END LOOP;
END;
/
```

USING NESTED TABLES

- ※ A nested **TABLE** contains a list of elements whose number can grow indefinitely.
- ※ To create an element, use the **EXTEND** method.

```
var.EXTEND;      -- Add an element to the collection
```

- **EXTEND** can add multiple elements:

```
var.EXTEND(30);
```

- ※ There are three ways to assign values to table elements:

- Use array-like syntax.

```
var(1) := value;
```

- You may assign one table variable to another of the same type.

```
other_var := var;
```

- Data may be **SELECTEd** or **FETCHed** into an indexed element of the table.

```
SELECT lastname  
      INTO roster(3)  
      FROM person WHERE id = 6399;
```

- ※ You can also create collections (**TABLEs** and **VARRAYs**) of **RECORDs**.
 - This allows your program to perform very complex or repeated processing of database data, without repeated retrievals.

You can pass a collection to a subprogram.

loop_stores.sql

```
DECLARE
    snum NUMBER := 1; -- Initialize store_number to 1
    maxsnum NUMBER;
    TYPE id_tab IS TABLE OF store.manager_id%TYPE; -- Definition
    mgr id_tab; -- Table variable declaration

    PROCEDURE calculate_bonuses (manager id_tab) IS
        rev NUMBER(9,2);
    BEGIN
        FOR s IN 1 .. manager.COUNT LOOP
            SELECT SUM(oi.quantity*p.price) INTO rev
            FROM order_item oi INNER JOIN product p USING (product_id)
                INNER JOIN order_header oh
                    USING (invoice_number)
            WHERE oh.store_number = s;
            UPDATE employee SET pay_amount = pay_amount + (rev * .05)
            WHERE id = manager(s);
        END LOOP;
    END;

BEGIN
    mgr := id_tab(); -- Table constructor
    SELECT max(store_number) INTO maxsnum FROM store;

    -- Load table with manager_ids:
    WHILE snum <= maxsnum
    LOOP
        mgr.EXTEND; -- Add an element to the table
        SELECT manager_id INTO mgr(snum)
        FROM store
        WHERE store_number = snum;
        snum := snum + 1;
    END LOOP;

    calculate_bonuses(mgr);

END;
/
```

USING VARRAYS

* A **VARRAY** contains a list of elements whose number has a fixed upper bound.

- **VARRAY** elements are consecutive; a **VARRAY** has no gaps or empty elements.
- The definition of your **VARRAY** type includes the upper bound.

```
TYPE emptab IS VARRAY(120) OF employee%ROWTYPE;
```

- This doesn't create the elements; you must use **EXTEND**.

- Attempting to **EXTEND** a **VARRAY** beyond its upper bound causes an error.

* The rules for creation and assignment of **VARRAY** elements, and of entire **VARRAYs**, are the same as those for **TABLEs**.

* You can't define **TABLEs** and **VARRAYs** of PL/SQL types, such as **BOOLEAN** or **PLS_INTEGER**, that have no corresponding Oracle type.

* You can initialize elements of a collection in the constructor, in the executable part of a block:

```
BEGIN
    empsal := salary_t(8.90, 8.50, 9.00);
    ...

```

... or in the declaration section:

```
DECLARE
    ...
    empsal salary_t := salary_t(8.90, 8.50, 9.00);
```

```
empvar.sql
```

```
DECLARE
    TYPE emp_var IS VARRAY(5) OF employee%ROWTYPE;
    emp emp_var := emp_var();
BEGIN
    emp.EXTEND;
    SELECT * INTO emp(1) FROM employee WHERE id = 7111;
    emp.EXTEND;
    SELECT * INTO emp(2) FROM employee WHERE id = 7881;

    emp.EXTEND;
    emp(3) := emp(2);
    emp(3).id := 10007;
    emp(3).store_number := 3;
    emp.EXTEND(2);

    emp.EXTEND; -- Error! Subscript outside of limit!
END;
/
```

COLLECTIONS IN DATABASE TABLES

- * You can use either a nested **TABLE** or a **VARRAY** as a column datatype in a database table.

- First, create a new **TYPE** in the database based on the collection.

```
CREATE TYPE email_type AS VARRAY(10) OF VARCHAR2(40);
```

- Second, create a table with this type for one of the columns.

```
CREATE TABLE emp_email
(
    id NUMBER,
    email email_type
);
```

- Each row in the new table will contain a collection of these values.

- * You could instead use a second table to hold the data in separate rows, if you want to avoid violating First Normal Form.

- First Normal Form states that a row's value for a column can only contain one data item.
 - To access data in a second table, you will need to use a **JOIN**.

- * Storing multiple values in a single row can simplify queries and reduce database storage.

get_emails.sql

```
SELECT *
  FROM emp_email;
```

- The **TABLE** function can be used to treat the column values like its own table of data.

table_query.sql

```
SELECT emp.id, mail.*
  FROM emp_email emp CROSS JOIN TABLE(email) mail;
```

- Using the **TABLE** function requires that you pass the name of a collection-based column from a table to the left within the **FROM** clause; this is called *left correlation*.

```
email_table.sql
CREATE TYPE email_type AS VARRAY(10) OF VARCHAR2(40)
/
CREATE TABLE emp_email
(
    id NUMBER,
    email email_type
)
/
INSERT INTO emp_email
(SELECT id,
     email_type(lastname || '.' || firstname || '@example.com',
                firstname || '.' || lastname || '@example.com')
  FROM employee JOIN person USING (id));
```

Data in a nested table column is stored separately from the main table data, in its own table. So, in essence, there is no reduced database storage. Varray data is either stored inline along with the table's other data, or as a LOB. But access to the data is always done through the main table, for either column type. Creating a table that uses a nested table requires an extra clause to specify a name for the underlying table that is created.

```
nested.sql
CREATE TYPE email_type AS TABLE OF VARCHAR2(40)
/
CREATE TABLE emp_email
(
    id NUMBER,
    email email_type
) NESTED TABLE email STORE AS emp_email_tab
/
INSERT INTO emp_email
(SELECT id,
     email_type(lastname || '.' || firstname || '@example.com',
                firstname || '.' || lastname || '@example.com')
  FROM employee JOIN person USING (id));
```

Information about nested tables that you have created can be found in the **USER_NESTED_TABLES** and **USER_NESTED_TABLE_COLS** Data Dictionary views.

```
query_nested.sql
SELECT * FROM user_nested_tables
/
SELECT table_name, column_name, data_type
  FROM user_nested_table_cols;
```

Information about varrays is kept in **USER_VARRAYS**.

```
query_varray.sql
SELECT *
  FROM user_varrays;
```

In SQL*Plus, you can also **DESCRIBE** both the table and the type itself.

```
DESC emp_email
```

```
DESC email_type
```

ASSOCIATIVE ARRAY COLLECTIONS

- ✳️ *Associative arrays* are collections that store their data as key-value pairs.
 - Valid key field types include **PLS_INTEGER**, **BINARY_INTEGER**, or **VARCHAR2** (or one of its subtypes, like **STRING**).
 - Each key must be unique from any other key inserted.
 - Using the same key as a prior insert replaces the previous value.
- ✳️ The key is similar to a primary key used in an Oracle table.
 - This makes associative arrays ideal for small lookup tables that can be populated each time the block of code is run or a package is referenced.
- ✳️ There are three steps to using an associative array:
 1. Define an associative collection type:

```
TYPE dept_count_type IS TABLE OF NUMBER INDEX BY VARCHAR2(30);
```

2. Declare a variable of that type:

```
dept_size dept_count_type;
```

3. Assign values into the collection:

```
dept_size('RD') := 45;  
dept_size('SALES') := 123;
```

- ✳️ No constructor notation is needed for associative arrays.
- ✳️ Unlike **VARARRAYs** and nested **TABLEs**, associative arrays cannot be stored in database columns.

```
assoc_arr.sql
VAR total NUMBER

DECLARE
    TYPE emptab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    emp_counts emptab;
    FUNCTION get_emp_counts RETURN emptab AS
        myemps emptab;
        cur_val NUMBER;
    BEGIN
        FOR s IN 1..7 LOOP
            SELECT COUNT(*)
            INTO cur_val
            FROM employee
            WHERE store_number = s;
            myemps(s) := cur_val;
        END LOOP;

        RETURN myemps;
    END get_emp_counts;

BEGIN
    :total := 0;
    emp_counts := get_emp_counts;
    FOR x IN 1..7 LOOP
        :total := :total + emp_counts(x);
    END LOOP;

END;
/
PRINT total
```

COLLECTION METHODS

- * Collection *methods* allow your program to examine collections.

| | |
|-------------------------|---|
| COUNT | Number of elements in a collection |
| LIMIT | Declared limit of a VARRAY ; NULL for a TABLE and associative arrays |
| FIRST, LAST | Indexes of the first and last elements |
| PRIOR(<i>n</i>) | Index of previous element before <i>n</i> |
| NEXT(<i>n</i>) | Index of next element after <i>n</i> |
| EXISTS(<i>n</i>) | TRUE if TABLE element <i>n</i> still exists |

- * Methods also allow your program to manipulate collections.

| | |
|---------------------------|---|
| EXTEND | Add an element to a collection (not used with associative arrays) |
| EXTEND(<i>n</i>) | Add <i>n</i> elements to a collection |
| TRIM | Remove last element from a collection |
| TRIM(<i>n</i>) | Remove last <i>n</i> elements from a collection |
| DELETE(<i>n</i>) | Delete TABLE element <i>n</i> or delete associative array element with key <i>n</i> |
| DELETE(<i>n,m</i>) | Delete TABLE elements <i>n</i> through <i>m</i> or delete associative array elements with key values between <i>n</i> and <i>m</i> |

- * An uninitialized collection is *atomically null* (it doesn't exist).

```
var := NULL; -- Reset var to be atomically null
```

- You must reinitialize a **NULL** collection by calling its constructor before using it.

```
var := vartype();
```

emptab.sql

```
DECLARE
    TYPE emp_tab IS TABLE OF employee%ROWTYPE;
    emp emp_tab := emp_tab();
BEGIN
    emp.EXTEND;
    SELECT * INTO emp(1) FROM employee WHERE id = 7111;
    emp.EXTEND;
    SELECT * INTO emp(2) FROM employee WHERE id = 7881;
    emp.EXTEND;
    emp(3) := emp(2);
    emp(3).id := 10007;
    emp(3).store_number := 3;
    emp.DELETE(2);
    emp.EXTEND;
    IF emp.EXISTS(2) THEN
        emp(4) := emp(2);
    ELSE
        emp(4) := emp(emp.NEXT(2));
    END IF;
END;
/
```

ITERATING THROUGH COLLECTIONS

- ※ Usually, you'll use a loop to iterate through the elements of a collection and perform some processing on each element.
- ※ To fill up a collection, you would use the **EXTEND** method while looping through your data set.
 - If you know the final size of the collection, use **EXTEND** just once before the loop and fill up the collection inside the loop.
- ※ To loop through an already filled collection, use **FIRST**, **NEXT**, and **LAST**.

```
i := list.FIRST;
WHILE i != list.LAST LOOP
    IF max < list(i) THEN
        max := list(i);
    END IF;
    i := list.NEXT(i);
END LOOP;
```

- If the indexes in your collection are sequential, you might only need to use **LAST** to find the end.

```
FOR i IN 1..list.LAST LOOP
    IF max < list(i) THEN
        max := list(i);
    END IF;
END LOOP;
```

varray.sql

```
DECLARE
    eid employee.id%TYPE;
    TYPE e_tab IS RECORD (
        id employee.id%TYPE,
        pay employee.pay_amount%TYPE,
        pay_type employee.pay_type_code%TYPE
    );
    erec e_tab;
    TYPE sal_tab IS VARRAY(200) OF e_tab;
    sal sal_tab := sal_tab();
    ecount NUMBER;
    num NUMBER := 1;
BEGIN
    SELECT count(*) INTO ecount FROM employee;
    SELECT min(id) INTO eid FROM employee;

    WHILE num <= ecount LOOP
        sal.EXTEND;
        SELECT id, pay_amount, pay_type_code INTO sal(num)
            FROM employee WHERE id = eid;
        SELECT min(id) INTO eid FROM employee WHERE id > eid;
        num := num + 1;
    END LOOP;

    FOR n IN 1..sal.LAST
    LOOP
        IF sal(n).pay_type = 'H' THEN
            sal(n).pay := sal(n).pay + .25;
        ELSIF sal(n).pay_type = 'C' THEN
            sal(n).pay := sal(n).pay * 1.05;
        END IF;
        UPDATE employee SET pay_amount = sal(n).pay
            WHERE id = sal(n).id;
    END LOOP;

END;
/
```

LABS

- ① In an anonymous block, create a collection (nested table or associative array) of records that will hold the cost of living index for several locations. Each record should contain fields for a city, state, and index value for that location. The cost of living index will be a percentage.

Use the following locations and percentages:

| | |
|-----|--------------------|
| 6% | Denver, CO |
| 52% | Los Angeles, CA |
| 12% | Las Vegas, NV |
| 23% | Seattle, WA |
| 12% | Reno, NV |
| 0% | Salt Lake City, UT |
| 76% | San Francisco, CA |

(Solution: *cost_of_living1.sql*)

- ② Modify the program from ① to loop through all elements in the collection and, for each one, perform an **UPDATE** to every employee that lives in that location. Give them a cost of living raise, increasing their **pay_amount** by the percentage for that location.

(Solution: *cost_of_living2.sql*)

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING

CHAPTER 12 - WORKING WITH LOBs

OBJECTIVES

- * Describe Oracle's implementation of Large Object datatypes.
- * Create and use **DIRECTORY** objects for files related to LOB programming.
- * Choose the appropriate LOB type for your data.
- * Work with both internal and external LOBs.
- * Use the API provided by **DBMS_LOB** to work with LOBs in PL/SQL.

LARGE OBJECT TYPES

- ※ Large Objects (LOBs) allow you to store very large items of unstructured data in the database.
- ※ There are three LOB types in Oracle:
 - **BLOB** — Binary data, such as images, videos, audio clips, or compiled programs.
 - **CLOB** and **NCLOB** — Text, such as XML documents or publications.
 - **BFILE** — A reference to a file outside of Oracle, on the host file system.
- ※ A single LOB value can be as large as 8 terabytes (or even 128 terabytes, depending on the storage parameters of a particular database.)
 - The **LONG RAW** and **LONG** datatypes, Oracle's original **BLOB** and **CLOB** implementations, hold up to 2 gigabytes.
 - Though still available (and used internally by Oracle), you should not use **LONG RAW** or **LONG** datatypes, and should convert legacy columns of these types to **BLOB** or **CLOB**.
- ※ Most of Oracle's character **VARCHAR2** functions (**SUBSTR**, **INSTR**, **||**, **regex** functions, etc.) work with **CLOBs**.

While there are some restrictions which distinguish LOBs from the other Oracle datatypes, LOBs are much less restricted than the older **LONG** types. For example:

- You can define any number of LOB columns in a database table.
- You can use a LOB as an attribute of a user-defined datatype.
- LOBs allow random (rather than sequential) access to the LOB data.
- You can access LOBs using normal SQL semantics.

Some restrictions for LOBs include:

- You cannot use a LOB as a primary key column.
- You cannot use a LOB column in an **ORDER BY** or **GROUP BY** clause.
- You cannot use **SELECT DISTINCT** on a LOB column.
- You cannot use a LOB column as an index key, or in a cluster.
- You cannot use a LOB column in the **FOR UPDATE OF** clause of an **UPDATE** trigger.

ORACLE DIRECTORIES

- * Oracle uses **DIRECTORY** objects to represent locations (directories or folders) in the host operating system's filesystem.
 - Locations for reading or writing files containing LOBs.
 - Locations for database export and import files, external tables and other OS files used by Oracle.
- * To create a **DIRECTORY**, first create and prepare the actual operating system directory to be represented.
 - Use **mkdir** (or some other utility) to create the directory itself.
 - Grant OS read and write permissions on the directory to the operating system user ID under which the Oracle database runs.
- * Use **CREATE DIRECTORY** to create the **DIRECTORY** object representing the operating system directory:

```
CREATE DIRECTORY tempfile_dir AS '/tmp';
```

- **DIRECTORY** objects are part of the **SYS** schema (that is, global to the database), not part of your own schema.
- You must have the **CREATE ANY DIRECTORY** system privilege to create a **DIRECTORY**.
- * Within Oracle, you can **GRANT** and **REVOKE** two privileges on a **DIRECTORY** object: **READ** and **WRITE**.

```
GRANT read,write ON tempfile_dir TO PUBLIC;
```

DIRECTORY objects are not specifically designed for or tied to LOBs, but since LOB programs often access external files (reading and writing **BLOBs** and **CLOBs**, accessing **BFILEs**, etc.) we usually have to set up a **DIRECTORY** for the LOB program.

Hands On:

Look up the defined directories on your system in the **ALL_DIRECTORIES** data dictionary view.

LOB LOCATORS

- ※ In a table, a LOB column actually stores not the LOB data, but a LOB locator.
 - The LOB locator is a reference, used internally by Oracle, to the actual LOB data.
- ※ A LOB column has one of three states:
 - **NULL** — The LOB locator itself is **NULL**, just like any other **NULL** value.
 - You can delete a LOB value by setting its column to **NULL**:

```
UPDATE table SET lobcolumn = NULL WHERE key = value;
```

- Both the locator and the corresponding LOB value are deleted.
 - You can't invoke LOB API functions on a **NULL** locator value.
- **EMPTY** — The column contains a LOB locator, which points to an empty (zero-length) LOB value.
 - When in this state, you can access and populate the LOB value.
 - Initialize a column to **EMPTY** with the **EMPTY_BLOB()** or **EMPTY_CLOB()** SQL function:

```
UPDATE table SET lobcolumn = EMPTY_CLOB() WHERE key = value;
```

- Populated — The column contains a LOB locator, which points to a non-empty LOB value.
- ※ You can store a LOB locator in a PL/SQL variable; however, you cannot use this stored locator beyond the current transaction or session.

INTERNAL LOBs

- ※ An *internal* LOB value is stored in Oracle's tablespaces.
- ※ The **BLOB**, **CLOB**, and **NCLOB** types are internal LOBs.
 - **BLOB** is for unstructured, binary data.
 - **CLOB** is for character data, and is stored in the database's character set.
 - **NCLOB** is for character data, stored in a (possibly variable-width) National Character Set.
- ※ The **BLOB**, **CLOB**, or **NCLOB** column stores a locator; the actual LOB data is stored either:
 - Inline — The LOB data is stored in the table's row, just like a **RAW** or **VARCHAR2** column value.
 - This only occurs when the LOB value is empty, or when the LOB value is less than 4Kb and **ENABLE STORAGE IN ROW** was specified for the LOB column.
 - Out-of-line — The out-of-line LOB value is stored separately from the other row data, possibly in a separate tablespace.
- ※ Internal LOB data participates fully in transactions — that is, they are subject to **COMMIT** and **ROLLBACK**.
- ※ Oracle implicitly converts between **NCLOB** and **CLOB** values, and between **CLOB** and **VARCHAR2**.
 - For example, you can fetch a **CLOB** value into a **VARCHAR2** PL/SQL variable.

warranty_search.sql

```
CREATE OR REPLACE FUNCTION warranty_search (pid VARCHAR2,
                                            searchfor VARCHAR2 )
  RETURN VARCHAR2
IS
  retval      VARCHAR2(80) := '';
  clobvalue   CLOB;
  foundtext   VARCHAR2(80) := '';
  len         NUMBER;
  ppos        NUMBER;
BEGIN
  -- Get the LOB locator for a warranty
  SELECT warranty_text INTO clobvalue FROM product
    WHERE product_id = pid;
  IF clobvalue IS NULL THEN
    retval := 'Not Found: Null warranty for product ID ' || pid ;
  ELSE
    len := length(clobvalue);
    IF len = 0 THEN
      retval := 'Not Found: Warranty empty for product ID ' || pid ;
    ELSE
      foundtext := REGEXP_SUBSTR(clobvalue, searchfor);
      IF (LENGTH(foundtext) > 0) THEN
        retval := 'Found: ' || SUBSTR(clobvalue,
                                       REGEXP_INSTR(clobvalue, searchfor) - 10,
                                       80 - LENGTH(foundtext) - 10 );
      ELSE
        retval := 'Not Found: "' || searchfor || '" in product '
                  || pid || ' warranty.';
      END IF;
    END IF;
  END IF;
  RETURN retval;
END;
/
SET SERVEROUTPUT ON
DECLARE
  found VARCHAR2(80);
BEGIN
  dbms_output.put_line('Searching for: ' || 'EXPRESS');
  found := warranty_search('CAFZ0000001', 'EXPRESS');
  dbms_output.put_line(found);
END;
/
```

EXTERNAL LOBs

- ※ The **BFILE** type is an external LOB.
 - An external LOB value is stored in host operating system's filesystem.
- ※ A **BFILE** column stores a locator; the actual LOB data is in an operating system file.
 - Any file can be referenced as a **BFILE**.
 - The Oracle Database server must have operating system permissions for accessing the file.
 - Access to **BFILE** data is read-only.
 - Deleting a **BFILE** value has no effect on the actual operating system file.
- ※ External LOB data does not participate in transactions.
- ※ Beginning in version 10g, Oracle uses **DIRECTORY** objects to represent parts of the operating system's filesystem.

```
CREATE DIRECTORY libdir AS '/home/app/lib';
```

- ※ Use **BFILERNAME** to obtain a **BFILE** reference to an external file:

```
UPDATE staff SET photo = BFILERNAME('LIBDIR', 'bob_dobbs.jpg')
WHERE id = 42;
```

To use a LOB locator for an external LOB (a **BFILE**), you must:

- Have an Oracle **DIRECTORY** object, representing an actual, physical directory path.
- The physical files to be represented by the **BFILEs** must exist.
- The operating system userid under which Oracle runs must have "read" permission on the files.

The following code example will create a **BFILE** on *hamlet.xml* and load it into an internal **LOB** inside an Oracle table.

lob_example.sql

```
DROP TABLE xmltable
/
CREATE TABLE xmltable (
    filename    VARCHAR2(20),
    xmlData     CLOB
)
/
CREATE OR REPLACE DIRECTORY TEMPFILE_DIR AS '/tmp'
/
CREATE OR REPLACE FUNCTION GetDoc( filename VARCHAR2 ) RETURN CLOB
IS
    file          BFILE;
    data          CLOB;
    src_offset    INTEGER := 1;
    dst_offset    INTEGER := 1;
    src_charset   NUMBER := NLS_CHARSET_ID('WE8MSWIN1252');
    lang_context  NUMBER := DBMS_LOB.DEFAULT_LANG_CTX;
    fileLen       INTEGER;
    warnings      INTEGER;
BEGIN
    file := BFILENAME('TEMPFILE_DIR',filename);
    DBMS_LOB.OPEN(file);
    DBMS_LOB.CREATETEMPORARY(data, TRUE, DBMS_LOB.SESSION);
    fileLen := DBMS_LOB.GETLENGTH(file);
    DBMS_LOB.LOADCLOBFROMFILE(data, file, fileLen, src_offset, dst_offset,
                               src_charset, lang_context, warnings);
    DBMS_LOB CLOSE(file);
    RETURN data;
END;
/
DECLARE
    clobData CLOB;
BEGIN
    clobData := GetDoc('hamlet.xml');
    INSERT INTO xmltable VALUES ('hamlet.xml',clobData);
END;
/
```

TEMPORARY LOBs

- ※ A temporary LOB is not part of a table or an object type.
 - It is stored in the database's temporary tablespace, and is not part of any schema.
- ※ A temporary LOB can be created:
 - Implicitly:

```
myData CLOB;
newData CLOB;
BEGIN
  SELECT warranty INTO myData FROM products WHERE ...
  -- myData now refers to a temporary LOB
  newData := myData;
  -- newData refers to a second temporary LOB
```

- Explicitly:

```
DBMS_LOB.CREATETEMPORARY(newData);
```

- ※ All temporary LOBs are deleted at the end of the session that created them.
 - Temporary LOBs created in PL/SQL by **SELECTing** or assigning to a LOB variable will be freed when the enclosing block terminates.
 - Your program can explicitly destroy any temporary LOBs it creates.

```
DBMS_LOB.FREETEMPORARY(myData);
```


THE DBMS_LOB PACKAGE

- ※ The **DBMS_LOB** package provides the PL/SQL programming interface for working with LOBs.
- ※ **DBMS_LOB** functions allow you to:
 - Open and read from **BFILEs**.
 - Examine the status and contents of a **BFILE**.
 - Load the contents of a **BFILE** into a **BLOB** or **CLOB** column.
 - Read the contents and properties of a LOB.
 - Modify the contents of a LOB.
 - Create and destroy temporary LOBs.
- ※ Many LOB operations can also use ordinary SQL semantics, and don't require **DBMS_LOB** procedures.
 - SQL semantics:

```
newData := myData;
newData := newData || blurb;
```

- **DMBS_LOB** semantics:

```
DBMS_LOB.COPY(newData, myData, DBMS_LOB.LOBMAXSIZE);
DBMS_LOB.WRITEAPPEND(newData, blurb, LENGTH(blurb));
```

Note:

When you **INSERT** or **UPDATE** to assign a LOB locator to a column, you often want to retain the value assigned for use in your program. Use PL/SQL's **RETURNING ... INTO** clause to capture the locator.

```
...
  clobvar      CLOB;
BEGIN
    UPDATE mytable SET clob_column = EMPTY_CLOB()
    WHERE keycol = somevalue
    RETURNING clob_column INTO clobvar;
...
-- do stuff with clobvar:
DBMS_LOB.LOADCLOBFROMFILE(clobvar, fileref, ...);
...
```

LABS

- ① In your host operating system's file system, identify the full path to the directory containing this chapter's example files (for example, *C:\o10plsql\ch11* or */home/student/o10pl/ch11*). Using OS commands, make sure this directory and all its parent directories are readable by the Oracle software account, and that the directory itself is writable by the Oracle software account.
(Solution: *pathname.txt*)
- ② Create an Oracle **DIRECTORY** object named **EXAMPLE_LOBDIR**, using the full path to the directory identified in ①.
(Solution: *crdirectory.sql*)
- ③ Create a PL/SQL procedure named **assign_warranty()**. It should take two arguments: a product ID and the name of a text file. It should open the named file in the **DIRECTORY** location you created in ②, and assign the contents of the text file as the **warranty_text** CLOB field for that product in the **product** table. Test your procedure by assigning the contents of *warranty3.txt* to product ID **SYBS0000006**.
(Solution: *assign_warranty.sql*)

INTRODUCTION TO ORACLE 10G PL/SQL PROGRAMMING
