Oracle 10g
PL/SQL Programming

Student Workbook

itcourseware
Training Materials For IT Professionals

ORACLE10G PL/SQL PROGRAMMING

Contributing Authors: Robert Freeman, Mary Gable, Danielle Hopkins, Julie Johnson, Roger Jones, Brian Peasland, 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 11

Course Objectives 12
Course Overview 14
Using the Workbook 15
Suggested References 16
Sample Database 18

Chapter 2 - Triggers 23

Beyond Declarative Integrity 24
Triggers 26
Types of Triggers 28
Row-Level Triggers 30
Trigger Conditions and Predicates 32
Cascading Triggers and Mutating Tables 34
Generating an Error 36
Triggers on Views 38
Schema and System Triggers 40
Maintaining Triggers 42
Labs 44
Labs (cont'd) 46

Chapter 3 - PL/SQL Variables and Datatypes 49

Declaring Variables 50
Datatypes 52
Subtypes 54
Character Data 56
Dates and Timestamps 58
Date Intervals 60
Anchored Types 62
Assignment and Conversions 64
Selecting into a Variable 66
Labs 68

Chapter 4 - PL/SQL Syntax and Logic 71

PL/SQL Blocks and Programs 72
Conditional Statements — IF/THEN 74
Conditional Statements — CASE 76
Comments and Labels 78
Loops 80
WHILE and FOR Loops 82
Labs 84

Chapter 5 - Stored Procedures and Functions 87

Stored Subprograms 88
Procedures and Functions 90
Creating a Stored Procedure 92
Calling a Stored Procedure 94
Passing Parameters and Default Arguments 96
Parameter Modes 98
Creating a Stored Function 100
Calling a Stored Function 102
Stored Functions and SQL 104
Local Procedures and Functions 106
Labs 108

Chapter 6 - Exception Handling 111

SQLCODE and SQLERRM 112
Exception Handlers 114
Nesting Blocks 116
Scope and Name Resolution 118
User-Defined Exceptions 120
Compile-Time Warnings 122
Labs 124

Chapter 7 - Records and Collections 127

Record Variables 128
Using the %ROWTYPE Attribute 130
VARRAY and Nested TABLE Collections 132
Using Nested TABLEs 134
Using VARRAYs 136
Associative Array Collections 138

 © 2011 ITCourseware, LLC

Collection Methods 140
Iterating Through Collections 142
FORALL Statement 144
BULK COLLECT Clause 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 - Using Packages 171

Packages 172
Oracle-Supplied Packages 174
The DBMS_OUTPUT Package 176
The DBMS_UTILITY Package 178
The UTL_FILE Package 180
The DBMS_METADATA Package 182
XML Packages 184
Networking Packages 186
Other Supplied Packages 188
Labs 190

Chapter 10 - Creating Packages 193

Structure of a Package 194
The Package Interface and Implementation 196
Package Variables 198
Overloading Package Functions and Procedures 200
Named Parameter Notation 202
REF CURSOR Variables 204
Labs 206

Chapter 11 - Working with LOBs 209

Large Object Types 210
Oracle Directories 212
LOB Locators 214
Internal LOBs 216
External LOBs 218
Temporary LOBs 220
The DBMS_LOB Package 222
Labs 224

Chapter 12 - Maintaining PL/SQL Code 227

Privileges for Stored Programs 228
PL/SQL Stored Program Compilation 230
The PL/SQL Execution Environment 232
Dependencies and Validation 234
Maintaining Stored Programs 236
Labs 238

Chapter 13 - The PL/SQL Execution Environment 241

The Server Process 242
PL/SQL Execution 244
The PL/SQL Compiler 246
Compiler Optimization 248
SQL — Parse 250
SQL — Execute and Fetch 252
Server Memory 254
Latches 256
Locks 258
Labs 260

Chapter 14 - Advanced Cursors 263

Cursor Types 264
Cursors and Storage 266
Spanning Commits Across FETCHes 268
Dynamic SQL in PL/SQL 270
Bulk Operations 272
Bulk Returns 274

 © 2011 ITCourseware, LLC

Limiting Results 276
Cursor Parameters 278
Cursor Variables 280
Strong and Weak Cursors 282
Using Cursor Variables 284
Cursor Type Errors 286
Cursor Subqueries 288
Labs 290

Chapter 15 - Dynamic SQL 293

Generating SQL at Runtime 294
Native Dynamic SQL vs. DBMS_SQL Package 296
The EXECUTE IMMEDIATE Statement 298
Using Bind Variables 300
Multi-row Dynamic Queries 302
Bulk Operations with Dynamic SQL 304
Using DBMS_SQL for DML and DDL 306
Using DBMS_SQL for Queries 308
Retrieving Meta Information with DBMS_SQL 310
Labs 312

Chapter 16 - Object-Oriented Oracle 315

Introducing Object-Oriented Oracle 316
Defining Object Types and Tables in SQL 318
Querying and Modifying Object Data 320
Object Methods 322
Inheritance 324
Type Evolution 326
Object Views 328
Object Types in PL/SQL 330
REF Pointers 332
Object Functions and Operators 334
Labs 336

Chapter 17 - Tuning PL/SQL 339

PL/SQL vs SQL 340
PL/SQL Performance Tips 342
Tuning Goals 344

Monitoring Wait Events 346
DBMS_PROFILER 348
DBMS_TRACE 350
Execution Plans 352
Interpreting Explain Plan Results 354
Execution Plan Details 356
Trace Files 358
TKPROF 360
Using trcsess 362
DBMS_APPLICATION_INFO 364
Labs 366

Chapter 18 - Debugging and Error Handling 369

Exception Management 370
Exception Propagation 372
User-Defined Exceptions 374
Exception Error Messages 376
Stack Management 378
Debugging with DBMS_OUTPUT 380
Debugging with a Table 382
Using UTL_FILE 384
Using DBMS_DEBUG 386
SQL Developer 388
Avoiding Bugs 390
Labs 392

Chapter 19 - Advanced Programming Topics 395

Autonomous Transactions 396
Invoker's Rights 398
Fine-Grained Access Control with DBMS_RLS 400
Creating Pipes with DBMS_PIPE 402
Writing to and Reading from a Pipe 404
Table Functions 406
Pipelined Table Functions 408
Enabling parallel execution 410
DETERMINISTIC Functions 412
Labs 414

 © 2011 ITCourseware, LLC

Chapter 20 - Interfacing With External Code 417

External Programs and Procedures 418
External Procedure Architecture 420
Configure Oracle For External Procedures 422
Creating a java Stored Procedure 424
Security and External Programs 426
The Job Scheduler 428
Manage and Drop External Jobs 430
Native Compilation of PL/SQL Code 432
The Oracle Call Interface (OCI and OCCI) 434
Pro*C and Pro*C++ 436
Using Pro*C and Pro*C++ 438
Perl DBI/DBD Architecture 440
Perl and Stored Procedures 442
ODBC 444
Using ODBC 446
JDBC 448

Chapter 21 - Working with XML 451

Databases and XML 452
Schema Validation 454
Unstructured and Structured Storage 456
The XMLType Datatype 458
XPath Expressions 460
Extracting XML Data 462
Generating XML 464
XMLQuery 466
XMLType Views 468
Oracle XML DB Repository 470
Labs 472

Appendix A - PL/SQL Versions, Datatypes and Language Limits 475

Appendix B - Oracle10g Supplied Packages 485

Solutions 493

Index 549

 © 2011 ITCourseware, LLC

CHAPTER 1 - COURSE INTRODUCTION

Course Objectives

- Create triggers on database tables.
- Use PL/SQL's datatypes for database and program data.
- Use program stucture 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.
- Use detailed understanding of the PL/SQL execution environment in your application design and tuning.
- Develop programs that make sophisticated and effective use of cursors.
- Use all kinds of dynamic SQL in your PL/SQL code.
- Design and write solutions using Oracle's object types.
- Use Oracle's tools and supplied packages to trace, profile, and tune your PL/SQL programs.

 © 2011 ITCourseware, LLC

- Use a variety of techniques and tools for debugging PL/SQL code.
- Write programs that interface between PL/SQL, and external procedures and programs.
- Use package state to solve application problems.
- Use autonomous transactions in stored subprograms and triggers.
- Choose which user's application context and rights will apply when a stored subprogram runs.
- Write high-performance code using **NOCOPY** and pipelined table functions.
- Create functions to implement fine-grained access control.
- Use **DBMS_PIPE** to set up inter-session communication between 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, SQL*Plus, and PL/SQL.

 © 2011 ITCourseware, LLC

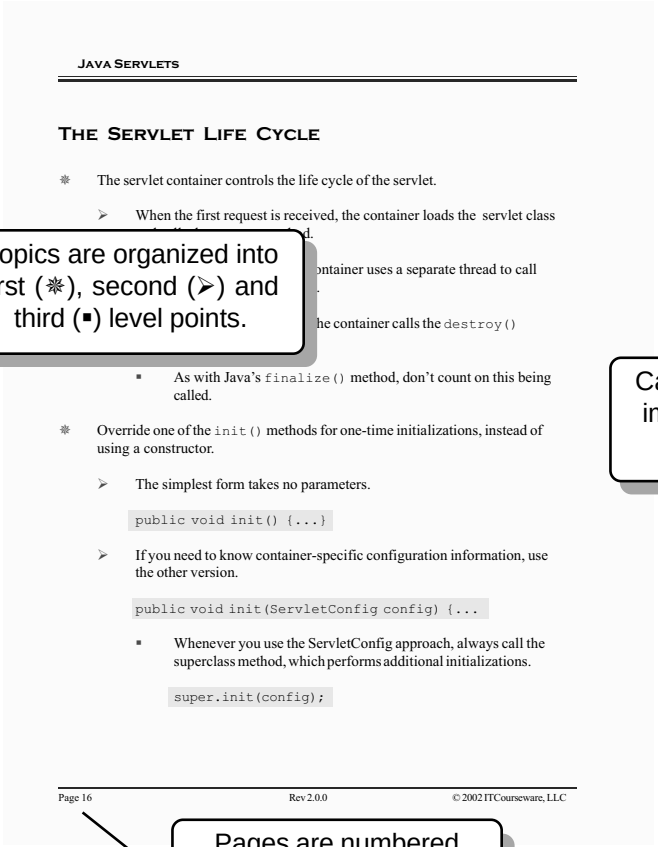
USING THE WORKBOOK

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

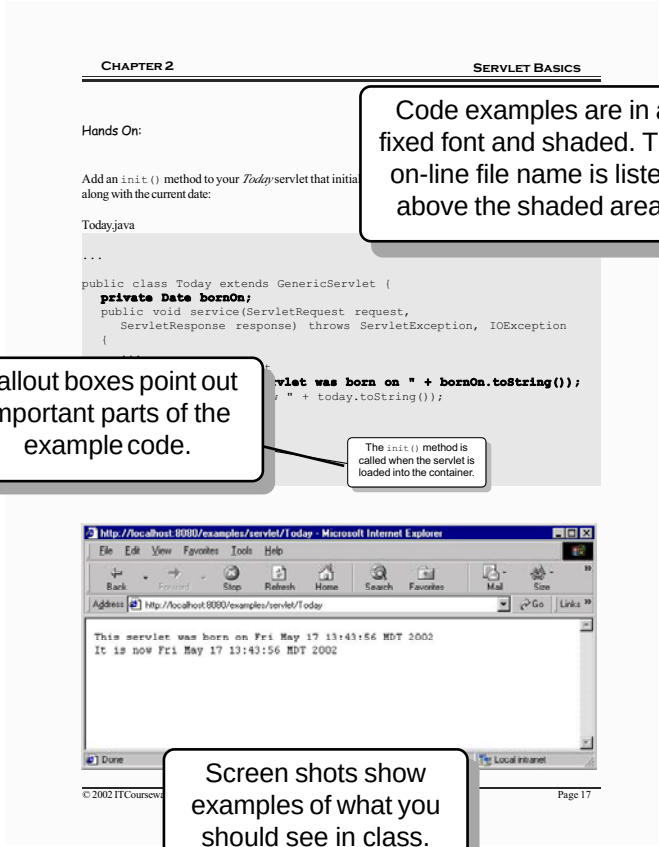
- The servlet container controls the life cycle of the servlet.
  - When the first request is received, the container loads the servlet class
  - ontainer uses a separate thread to call
  - he container calls the `destroy()`
    - As with Java's `finalize()` method, don't count on this being called.
- 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);`

Page 16 Rev 2.0.0 © 2002 ITCourseware, LLC


CHAPTER 2 SERVLET BASICS

Hands On:

Add an `init()` method to your *Today* servlet that initia
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
  {
```

```
vlet was born on " + bornOn.toString());
" + today.toString());
```

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

© 2002 ITCoursew Page 17

Topics are organized into first (❋), second (➢) and third (▪) level points.

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.

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. 2001. ***Oracle PL/SQL Best Practices***. O'Reilly and Associates, Sebastopol, CA. ISBN 0596001215

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

Feuerstein, Steven. 2002. ***Oracle PL/SQL Programming***. O'Reilly and Associates, Sebastopol, CA. ISBN 0596003811

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. 2003. ***Oracle PL/SQL by Example***. Prentice Hall PTR, Upper Saddle River, NJ. ISBN 0131172611

 © 2011 ITCourseware, LLC

Trezzo, Joseph C., Bradley D. Brown, and Richard J. Niemiec. 1999. ***Oracle PL/SQL Tips and Tricks***. McGraw-Hill Osborne, Emeryville, CA. ISBN 0078824389

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

Urman, Scott and Tim Smith. 1996. ***Oracle PL/SQL Programming***. Oracle Press, Emeryville, CA. ISBN 0078821762

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

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 employees'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.

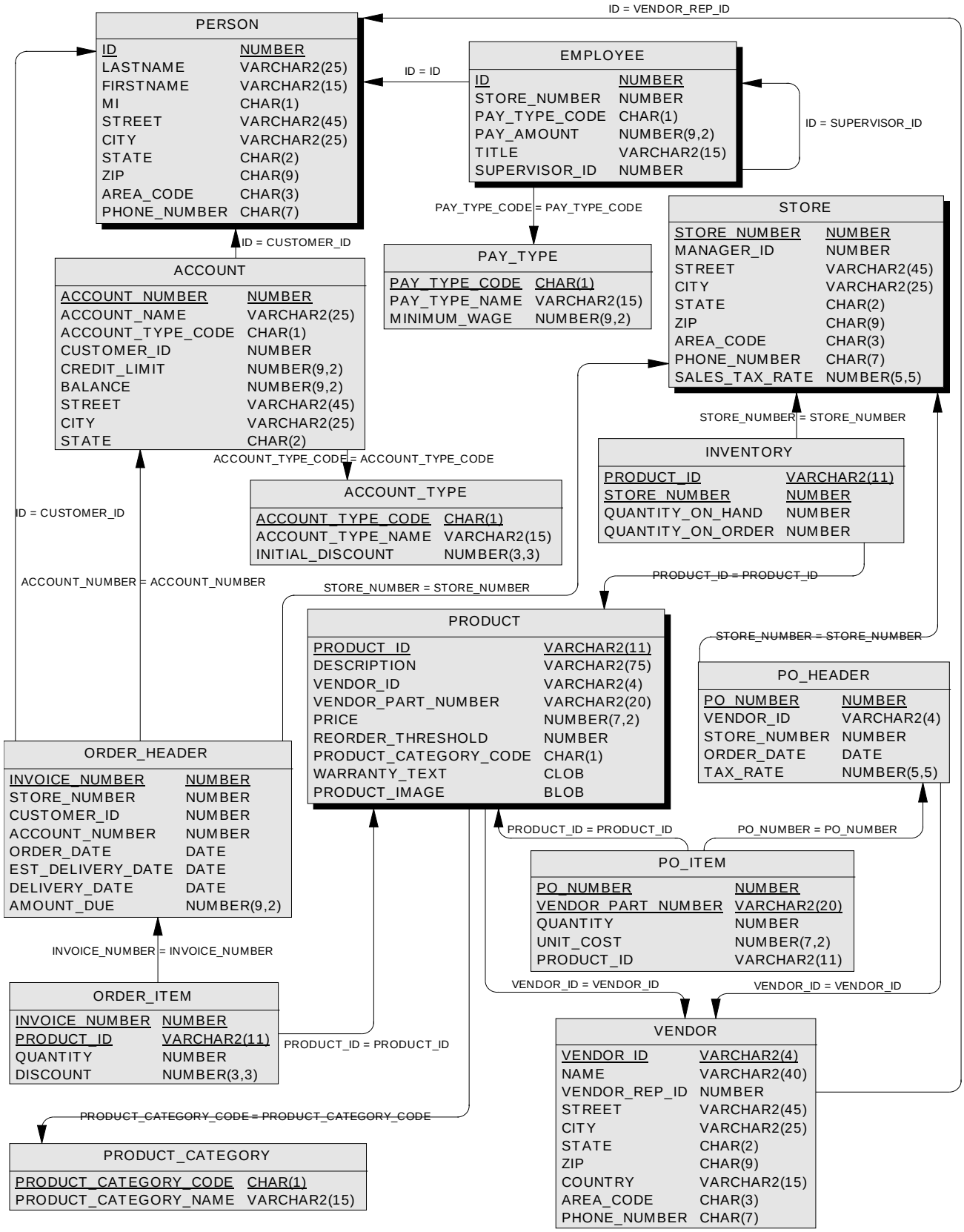
Each product we sell has a product ID and description. In addition, we keep track of the vendor we purchase that product from, the ID for that product, and the category ("Software," "Peripheral," or "Service"). We also store the address, phone, and sales rep ID for each individual vendor.

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.

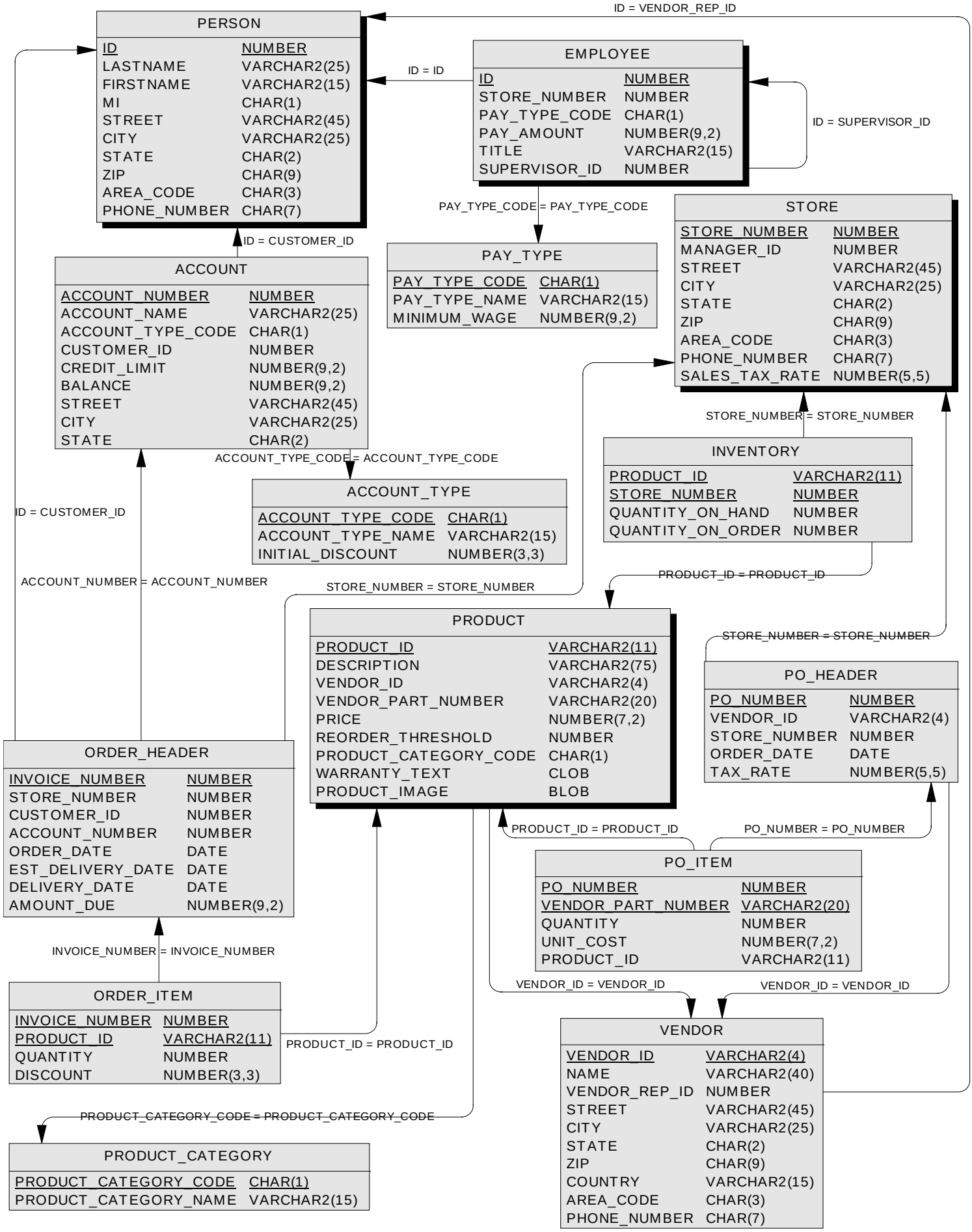
We maintain the address, phone number, and manager ID for each store. Each store has a unique store number. We record the sales tax rate for that store.

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.

 © 2011 ITCourseware, LLC


PERSON
ID NUMBER
LASTNAME VARCHAR2(25)
FIRSTNAME VARCHAR2(15)
MI CHAR(1)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
ID = VENDOR_REP_ID
ID = ID
EMPLOYEE
ID NUMBER
STORE_NUMBER NUMBER
PAY_TYPE_CODE CHAR(1)
PAY_AMOUNT NUMBER(9,2)
TITLE VARCHAR2(15)
SUPERVISOR_ID NUMBER
ID = SUPERVISOR_ID
PAY_TYPE_CODE = PAY_TYPE_CODE
STORE
STORE_NUMBER NUMBER
MANAGER_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
SALES_TAX_RATE NUMBER(5,5)
ID = CUSTOMER_ID
PAY_TYPE
PAY_TYPE_CODE CHAR(1)
PAY_TYPE_NAME VARCHAR2(15)
MINIMUM_WAGE NUMBER(9,2)
ACCOUNT
ACCOUNT_NUMBER NUMBER
ACCOUNT_NAME VARCHAR2(25)
ACCOUNT_TYPE_CODE CHAR(1)
CUSTOMER_ID NUMBER
CREDIT_LIMIT NUMBER(9,2)
BALANCE NUMBER(9,2)
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
STORE_NUMBER = STORE_NUMBER
INVENTORY
PRODUCT_ID VARCHAR2(11)
STORE_NUMBER NUMBER
QUANTITY_ON_HAND NUMBER
QUANTITY_ON_ORDER NUMBER
ACCOUNT_TYPE_CODE = ACCOUNT_TYPE_CODE
ACCOUNT_TYPE
ACCOUNT_TYPE_CODE CHAR(1)
ACCOUNT_TYPE_NAME VARCHAR2(15)
INITIAL_DISCOUNT NUMBER(3,3)
ID = CUSTOMER_ID
ACCOUNT_NUMBER = ACCOUNT_NUMBER
STORE_NUMBER = STORE_NUMBER
PRODUCT_ID = PRODUCT_ID
PRODUCT
PRODUCT_ID VARCHAR2(11)
DESCRIPTION VARCHAR2(75)
VENDOR_ID VARCHAR2(4)
VENDOR_PART_NUMBER VARCHAR2(20)
PRICE NUMBER(7,2)
REORDER_THRESHOLD NUMBER
PRODUCT_CATEGORY_CODE CHAR(1)
WARRANTY_TEXT CLOB
PRODUCT_IMAGE BLOB
STORE_NUMBER = STORE_NUMBER
PO_HEADER
PO_NUMBER NUMBER
VENDOR_ID VARCHAR2(4)
STORE_NUMBER NUMBER
ORDER_DATE DATE
TAX_RATE NUMBER(5,5)
ORDER_HEADER
INVOICE_NUMBER NUMBER
STORE_NUMBER NUMBER
CUSTOMER_ID NUMBER
ACCOUNT_NUMBER NUMBER
ORDER_DATE DATE
EST_DELIVERY_DATE DATE
DELIVERY_DATE DATE
AMOUNT_DUE NUMBER(9,2)
PRODUCT_ID = PRODUCT_ID
PO_NUMBER = PO_NUMBER
PO_ITEM
PO_NUMBER NUMBER
VENDOR_PART_NUMBER VARCHAR2(20)
QUANTITY NUMBER
UNIT_COST NUMBER(7,2)
PRODUCT_ID VARCHAR2(11)
INVOICE_NUMBER = INVOICE_NUMBER
ORDER_ITEM
INVOICE_NUMBER NUMBER
PRODUCT_ID VARCHAR2(11)
QUANTITY NUMBER
DISCOUNT NUMBER(3,3)
PRODUCT_ID = PRODUCT_ID
VENDOR_ID = VENDOR_ID
VENDOR_ID = VENDOR_ID
VENDOR
VENDOR_ID VARCHAR2(4)
NAME VARCHAR2(40)
VENDOR_REP_ID NUMBER
STREET VARCHAR2(45)
CITY VARCHAR2(25)
STATE CHAR(2)
ZIP CHAR(9)
COUNTRY VARCHAR2(15)
AREA_CODE CHAR(3)
PHONE_NUMBER CHAR(7)
PRODUCT_CATEGORY_CODE = PRODUCT_CATEGORY_CODE
PRODUCT_CATEGORY
PRODUCT_CATEGORY_CODE CHAR(1)
PRODUCT_CATEGORY_NAME VARCHAR2(15)

 © 2005 ITCourseware, LLC



© 2005 ITCourseware, LLC

 © 2011 ITCourseware, LLC

CHAPTER 2 - TRIGGERS

OBJECTIVES

- Distinguish between declarative and non-declarative integrity contraints.
- Create triggers to perform customized auditing and logging, special data validation, etc.
- Choose the correct type of trigger to fit your purpose.
- Avoid creating triggers that will cause mutating table errors.
- Use a trigger to raise an error when a statement violates your customized integrity constraints.
- Create effective triggers for updating views.
- Create triggers fired by database activities or DDL.
- Develop and maintain triggers.

BEYOND DECLARATIVE INTEGRITY

- *Declarative integrity* is defined by:
  - Primary and unique keys
  - Check constraints
  - Foreign keys
- Declarative integrity can enforce only simple, static conditions: uniqueness, existence of a parent record, and column values in a predefined range or set.
  - Oracle automatically enforces these constraints for all DML statements.
- A *trigger* allows you to define flexible, customized, dynamic constraints on any table or view, for any DML action.
  - Oracle fires your trigger any time the specified DML statement is executed on the trigger's table or view.
- There are three types of triggers:
  - Table triggers react to DML actions on a table.
  - DDL triggers are fired by schema events, like **CREATE** or **ALTER**.
  - System triggers react to database events.
- Use triggers to:
  - Enforce business rules in your database.
  - Implement complex referential or other integrity rules.
  - Generate keys or other column values for new rows.
  - Implement auditing, security, or logging.
- Don't use triggers to apply rules that you can enforce using normal declarative integrity constraints.

 © 2011 ITCourseware, LLC

Hands On:
How can you find an employee's current salary?

```
SELECT id, pay_amount
  FROM employee
 WHERE id = some_employee_id;
```

How can you increase an employee's pay by 10%?

```
UPDATE employee
   SET pay_amount = pay_amount * 1.1
 WHERE id = some_employee_id;
```

How can you find the last time an employee's pay was increased, including the old and new amount, when it was changed, and who changed it?

Hmm.

Fine-grained auditing, using the **DBMS_FGA** package, allows you to audit DML activity on specific tables and columns. However, it doesn't allow you to retrieve the before and after column values for each row. To satisfy this logging requirement, we can create a database table and insert a log record each time any employee's pay amount is updated.

Rather than requiring each user and application to log such changes themselves, we can use a trigger to automate this logging.

Let's define a database table to store the logging information. What columns do we need?

TRIGGERS

- A *trigger* is a PL/SQL block stored in the database that Oracle executes automatically under certain conditions.

- Each table trigger has a name, which must be distinct within your schema.

```
CREATE [OR REPLACE] TRIGGER trigger_name
       {BEFORE | AFTER}
       {INSERT | UPDATE [OF col_name [,col_name,...]] |
       DELETE}
       [OR {INSERT | UPDATE [OF col_name...]|DELETE}...]
ON table_name
  [FOR EACH ROW]
  [WHEN (condition)]
DECLARE
  declarations
BEGIN
  statements
END;
/
```

- An **INSERT**, **UPDATE**, or **DELETE** statement can fire a table trigger.
  - A trigger may fire for just one kind of DML action:

```
CREATE OR REPLACE TRIGGER emp_change_trigger
AFTER UPDATE OF employee
```

  - Or for more than one kind of DML action:

```
CREATE OR REPLACE TRIGGER person_log
AFTER INSERT OR UPDATE OR DELETE OF person
```

 © 2011 ITCourseware, LLC

Hands On:

Let's suppose we came up with the following table for logging pay changes in the **employee** table:

```
CREATE TABLE pay_change_log
(
  id NUMBER,
  old_amount  NUMBER(9,2),
  new_amount  NUMBER(9,2),
  date_changed DATE,
  changed_by  VARCHAR2(30)
);
```

Note:

The **id** column is for the ID of the employee whose pay was changed. The **changed_by** column will be the Oracle user who made the change; usernames are standard Oracle identifiers, so we use **VARCHAR2(30)**. This table is not part of our relational schema and doesn't need a primary key or other constraints, though we certainly could add a serial number or some other key field if we wanted to.

Go ahead and create this table now.

Now we need to define a trigger that will add a record to our new **pay_change_log** table any time an employee's pay amount changes.

Which table should our trigger be associated with?

Which DML statement or statements should fire our trigger?

Should it fire before or after the statement occurs?

Should the trigger fire only one time when a DML statement is executed, or repeatedly, once for each row affected by the statement?

How will the trigger determine the values to insert into the log table?

TYPES OF TRIGGERS

- You must specify when the table trigger is to fire:
  - **BEFORE** — the trigger executes before the specified DML action occurs.
    - The trigger can reject the action or change its effects.
  - **AFTER** — the trigger executes after the specified action occurs.
    - The trigger can log or audit activities, or collect statistics.
- By default, a trigger will fire once for the specified DML operation.
  - This is called a *statement-level trigger*.
- You can define the trigger to fire once **FOR EACH ROW** affected by the DML operation.
  - This is called a *row-level trigger*.
  - Remember that a single **INSERT**, **UPDATE**, or **DELETE** statement can affect multiple rows.
- A table can have several triggers of each kind; they fire in this order:

  1. All statement-level **BEFORE** triggers.
  2. All row-level **BEFORE** triggers.
  3. All row-level **AFTER** triggers.
  4. All statement-level **AFTER** triggers.

  - If you have more than one trigger of a given type, then the triggers of that type will fire in an unspecified order.

 © 2011 ITCourseware, LLC

Hands On:
Let's write the specification for our trigger, then fill in the body.

```
CREATE OR REPLACE TRIGGER pay_change
 AFTER UPDATE OF pay_amount
    ON employee
   FOR EACH ROW
DECLARE
  chdate DATE;
  chuser VARCHAR2(30);
BEGIN
END;
```

Note:
AFTER triggers typically perform logging, whereas **BEFORE** triggers usually validate data and check custom integrity constraints. We use **CREATE OR REPLACE**, so that as we edit and rerun our **CREATE TRIGGER** statement, we won't have to explicitly drop the trigger first.

As we'll soon see in greater detail, a PL/SQL program such as a trigger can declare program variables for its own internal use. All such declarations must be placed in a **DECLARE** section at the top of the program body.

How can we get the current date and time?

How about the Oracle username of the person executing the DML statement?

```
CREATE OR REPLACE TRIGGER pay_change
  ... -- other code omitted
BEGIN
  chdate := SYSDATE;
  chuser := USER;
END;
/
```

Note:
The **USER** function returns the Oracle user who executes the DML statement. The assignment token in PL/SQL is **:=** .

Now we just need the employee's ID number and the old and new pay amounts.

ROW-LEVEL TRIGGERS

- A row-level trigger can access the data affected by the triggering DML statement.
- PL/SQL predefines two variables in row-level triggers.
  - **:old** — defined in **UPDATE** and **DELETE** row-level triggers
  - **:new** — defined in **INSERT** and **UPDATE** row-level triggers
  - These pseudo-record variables have fields corresponding to the columns of the triggering table.

```
:old.pay_amount
```

- A **BEFORE INSERT** or **BEFORE UPDATE** trigger can modify the values about to go into the database by simply assigning values to the fields in the **:new** record.

```
:new.pay_amount := 9.50;
```

  - Subsequent row-level triggers (if you have multiple row-level triggers defined for the action) will see the changes made to **:new** by previously-fired triggers.

Trigger	:old	:new
INSERT	**NULL**	Values being inserted.
UPDATE	Original column values before the statement.	New column values in the **UPDATE** statement.
DELETE	Column values in the deleted row.	**NULL**

 © 2011 ITCourseware, LLC

Hands On:
Our trigger is almost finished. We just need to insert the data into the log table.

```
CREATE OR REPLACE TRIGGER pay_change
 AFTER UPDATE OF pay_amount
    ON employee
   FOR EACH ROW
DECLARE
  chuser VARCHAR2(30);
  chdate DATE;
BEGIN
  chuser := USER;     -- The Oracle user updating the table.
  chdate := SYSDATE;  -- Time of trigger firing.

  INSERT INTO pay_change_log
  VALUES ( :old.id, :old.pay_amount, :new.pay_amount, chdate,
     chuser );
END;
/
```

Note:
In a PL/SQL program, any text following a **--** is a comment. Observe that each PL/SQL program statement is terminated by a semicolon, **;**, including the SQL **INSERT** statement and the program itself (**END;**).

That should do it. Create this trigger now. If you see:

```
Warning: Trigger created with compilation errors.
```

then you've probably made a simple typo. The instructor will help you work out any syntax problems so your trigger compiles. Start by looking up error messages in the USER_ERRORS Data Dictionary table:

```
SELECT text FROM user_errors WHERE name = 'PAY_CHANGE';
```

SQL*Plus provides a shorthand for this command:

```
SQL> SHOW ERRORS
```

Once you've created it, how can you test your trigger?

TRIGGER CONDITIONS AND PREDICATES

- The optional **WHEN** clause determines when a row-level trigger will fire.
  - The condition is tested for each row, and the trigger fires if the condition is **TRUE** for the row.
  - You can reference the **new** (for **INSERT** and **UPDATE** triggers) and **old** (for **UPDATE** and **DELETE** triggers) pseudo-variables in the **WHEN** clause expression.
  - In the **WHEN** clause, **old** and **new** are not considered bind variables; you don't prefix them with **:** in the **WHEN** clause.
- For an **UPDATE** trigger, you can restrict firing of the trigger to statements that affect a specific column or columns.

```
... BEFORE UPDATE OF pay_amount, pay_type_code ON employee ...
```

  - An **UPDATE** affecting any of the listed columns will fire the trigger.
- In a row-level trigger fired by multiple DML actions (such as a trigger **ON INSERT OR UPDATE**), you can use predicates to determine which action fired the trigger at runtime.
  - The predicates, **INSERTING**, **UPDATING**, and **DELETING**, evaluate to **TRUE** or **FALSE**, depending on which DML statement fired the trigger.

```
IF INSERTING THEN
   :new.last_update_date = SYSDATE;
END IF;
```

  - In an **UPDATE** trigger, you can determine if a specific column is being updated by the current statement:

```
IF UPDATING ('PAY_AMOUNT') THEN ...
```

 © 2011 ITCourseware, LLC

Let's create a trigger called **minimum_hourly_wage**:

min_hourly_wage.sql

```
CREATE OR REPLACE TRIGGER minimum_hourly_wage
  BEFORE INSERT OR UPDATE OF pay_amount
      ON employee
     FOR EACH ROW
    WHEN (new.pay_type_code = 'H')
DECLARE
   min_wage NUMBER(9,2);
BEGIN
   SELECT minimum_wage INTO min_wage
     FROM pay_type WHERE pay_type_code = 'H';
   IF :new.pay_amount < min_wage THEN
      :new.pay_amount := min_wage;
   END IF;
END;
/
```

In the **WHEN** clause, refer to the old and new records as old and new, not **:old** and **:new**.

Note:
To rereive data into a variable using a SQL query, add the clause **INTO** variablename after the **SELECT** clause:

```
SELECT minimum_wage
  INTO min_wage
  FROM pay_type
 WHERE pay_type_code = 'H';
```

You'll need the ID of an hourly employee to test this trigger. Does the trigger fire when you update a temporary (**pay_type_code = 'T'**) employeee's pay? In the **WHEN** clause, refer to the old and new records as **old** and **new**, not **:old** and **:new**.

Cascading Triggers and Mutating Tables

- A *cascading trigger* is a trigger which executes a DML statement on another table, which then fires any triggers on that table.

- A *mutating table* is a table which a DML statement is modifying, or a table that might be affected by a **DELETE CASCADE** constraint.

  - Row level triggers can't use DML to modify, or even select from, a mutating table.

    - An **INSERT** trigger can't itself perform an **INSERT** on its triggering table.

    - Calculating a new primary key value from a mutating table is illegal.

- If a mutating table error occurs, it will happen when the triggering statement is executed.

  - Oracle reverses the effects of the triggering statement and trigger body, and the user or application gains control again.

 © 2011 ITCourseware, LLC

cascade.sql

```
CREATE OR REPLACE TRIGGER person_id_trig
   AFTER UPDATE OF id ON person
   FOR EACH ROW
DECLARE
BEGIN
   UPDATE employee
      SET id = :new.id
    WHERE id = :old.id;
   ... -- UPDATE account, order_header, and vendor
END;
/
CREATE OR REPLACE TRIGGER employee_id_trig
   AFTER UPDATE OF id ON employee
   FOR EACH ROW
DECLARE
BEGIN
   UPDATE store
      SET manager_id = :new.id
    WHERE manager_id = :old.id;
/*
The following update won't work, because
we are trying to update a mutating table!
   UPDATE employee
      SET supervisor_id = :new.id
    WHERE supervisor_id = :old.id;
*/
END;
/
```

Notice in the first trigger, **person_id_trig**, that we update an employee's id when updating its corresponding person id. The person table is the mutating table. The second trigger, **emp_id_trig**, responds to the update in the first trigger, updating the store manager's id when necessary. This is known as *cascading triggers*. If we attempt to update a supervisor's id in the second trigger, however, we get a runtime error, because we are attempting to update the mutating table (**employee**).

Hands On:

Suppose we want to create another trigger on the **employee** table, so that the pay amount for hourly workers can't be set below the minimum wage. If someone tries to violate this business rule, our trigger will raise an error.

Which DML statement or statements should fire our trigger?

Should it fire before or after the statement occurs?

Should this be a row-level or statement-level trigger?

GENERATING AN ERROR

- Use **raise_application_error** to make our trigger generate an error.

```
raise_application_error(errcode, errmsg, [keep_error_stack])
```

  - The first argument, ***errcode***, must be a number between -20000 and -20999, which Oracle sets aside for custom errors.
  - The second argument, ***errmsg***, is a character string up to 2048 bytes, which describes why the error occurred.
  - An optional third argument ***keep_error_stack*** is a boolean value.
    - **TRUE** means add the error to any other errors on the stack.
    - **FALSE** (default) means replace existing errors with this error.

- If a DML statement fires a trigger, and the trigger raises an error, the user will see the error code and message, and the triggering statement will fail.
  - If any one of a series of cascading triggers causes an error, none of the actions made by any of the triggers will take effect.

- You cannot use **COMMIT** or **ROLLBACK** inside a trigger, because your trigger fires within the context of a single DML statement which is part of the current transaction.
  - If the user rolls back after the trigger completes, then any DML statements executed by the trigger will also roll back.

 © 2011 ITCourseware, LLC

Hands On:

Let's modify our **minimum_hourly_wage** trigger, and have it generate an error, rather than just modify the invalid value:

min_hourly_wage_err.sql:

```
CREATE OR REPLACE TRIGGER minimum_hourly_wage
  BEFORE INSERT OR UPDATE OF pay_amount
      ON employee
     FOR EACH ROW
    WHEN (new.pay_type_code = 'H')
DECLARE
   min_wage NUMBER(9,2);
BEGIN
   SELECT minimum_wage INTO min_wage
     FROM pay_type WHERE pay_type_code = 'H';
   IF :new.pay_amount < min_wage THEN
      raise_application_error(-20002,
        'Hourly pay must meet or exceed minimum wage, ' || min_wage);
   END IF;
END;
/
```

Triggers on Views

- An **INSTEAD OF** trigger lets you modify a non-updatable view.
  - Oracle executes your trigger **INSTEAD OF** the DML statement on the view.
  - If you create an **INSTEAD OF** trigger on an updatable view, Oracle will still execute the trigger instead of your DML statement.
- A view is non-updatable if it contains:
  - A join on 2 or more tables (there are many exceptions to this rule).
  - A set or **DISTINCT** operator.
  - An aggregate or analytical function.
  - A subquery in a **SELECT** list or a subquery that is **READ ONLY**.
  - A **GROUP BY**, **MODEL**, **CONNECT BY**, **START WITH**, or **ORDER BY** clause.
  - A collection expression (i.e., concatenating values) in a **SELECT** list.
- The restrictions on **INSTEAD OF** triggers are:
  - You may only apply them to views, not tables!
  - The **:old** and **:new** values are read-only.
  - An **INSTEAD OF UPDATE** trigger may not use the **[OF *columns*]** option.

 © 2011 ITCourseware, LLC

instead_of.sql

```
DROP VIEW name_view
/

CREATE VIEW name_view (id, name) AS
SELECT id, lastname || ',' || firstname
  FROM person
 ORDER BY 1
/

CREATE OR REPLACE TRIGGER name_trig
   INSTEAD OF INSERT ON name_view
   FOR EACH ROW
DECLARE
   last VARCHAR2(25);
   first VARCHAR2(15);
   comma_pos NUMBER;
BEGIN
   comma_pos := INSTR(:new.name, ',');
   last := SUBSTR(:new.name, 1, comma_pos - 1);
   first := SUBSTR(:new.name, comma_pos + 1);
   -- Insert record into the base table PERSON
   INSERT INTO person (id, lastname, firstname)
          VALUES (:new.id, last, first);
END;
/

INSERT INTO name_view
       VALUES (person_id_seq.nextval, 'Troy,Diana')
/
```

SCHEMA AND SYSTEM TRIGGERS

- You can make system triggers fire when certain database or schema (client) events occur.

```
CREATE OR REPLACE TRIGGER triggername
   {BEFORE | AFTER} {event(s)}
   ON {SCHEMA | DATABASE}
...
```

  - Schema events include DDL and certain other statements for a schema.
    - They pertain to any objects in a particular schema; you cannot specify individual objects.
  - Database events include instance **STARTUP** and **SHUTDOWN**.
    - You must have **ADMINISTER DATABASE TRIGGER** system privilege to create a system event trigger.
  - You may associate **SERVERERROR**, **LOGON** and **LOGOFF** triggers with either the database instance or a schema.

- Use Oracle's built-in event functions to retrieve information about the triggering event.

ddl_event.sql

```
...
CREATE OR REPLACE TRIGGER ddl_trig
  BEFORE CREATE OR ALTER
  ON SCHEMA
BEGIN
   INSERT INTO ddl_events
          VALUES (ora_dict_obj_owner, ora_dict_obj_name,
                  ora_dict_obj_type, ora_login_user, sysdate);
END;
/
```

 © 2011 ITCourseware, LLC

Oracle provides many built-in functions for events. Examples include:

Event	Description	Built-in Functions	
CREATE	When an object is created.	ora_sysevent ora_login_user ora_instance_num ora_database_name	ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
	When a table is created.	ora_is_creating_nested_table	
ALTER	When an object is altered.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type	ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_passwo-rd ora_is_alter_column ora_is_drop_column
DROP	When an object is dropped.	ora_sysevent ora_login_user ora_instance_num ora_database_name	ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
GRANT	When a grant statement is issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name	ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privileges
DDL	When most SQL DDL statements are issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name	ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
RENAME	When a rename statement is issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name	ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type
AFTER LOGON BEFORE LOGOFF	At the start of a user logon or logoff.	ora_sysevent ora_login_user	ora_instance_num ora_database_name

MAINTAINING TRIGGERS

- To modify a trigger, you must first **DROP TRIGGER *triggername***, then **CREATE TRIGGER *triggername***, using your modified trigger definition.
  - **CREATE OR REPLACE TRIGGER *triggername*** drops the old trigger for you, saving you a step.
- When you create a trigger, Oracle:
  - Checks the syntax of your trigger for language errors.
  - Resolves the names of any tables, columns, or other objects your trigger references.
  - Compiles the trigger into efficient, binary-executable code (M Code).
  - Stores both the source code and M Code in the system catalog.
  - Enables your trigger.
- You can use **ALTER TRIGGER** and **ALTER TABLE** to disable and enable triggers:

```
ALTER TRIGGER trigger_name DISABLE;

ALTER TRIGGER trigger_name ENABLE;

ALTER TABLE table_name ENABLE ALL TRIGGERS;
```

- If you alter or drop an object referenced by your trigger, Oracle disables your trigger and marks it as invalid.
  - Compile the trigger to re-validate it; fix any problems that are found.

```
ALTER TRIGGER trigger_name COMPILE;
```

 © 2011 ITCourseware, LLC

Normally, you compose your trigger in a text file, and then run the file as a script. That makes it easy to modify the trigger as your needs change:

```
CREATE OR REPLACE TRIGGER minimum_hourly_wage
   BEFORE INSERT OR UPDATE OF pay_amount
       ON employee
      FOR EACH ROW
     WHEN (new.pay_type_code IN ('H','T'))
DECLARE
   ...
```

If you lose the file containing the source code for your trigger, don't worry, you can always retrieve your source code from the system catalog:

```
DESCRIBE user_triggers
```

Note that the **TRIGGER_BODY** column of **USER_TRIGGERS** is a **LONG**. SQL*Plus usually truncates **LONG** values that you select (since a **LONG** value might be up to 2 Gigabytes). To retrieve more than SQL*Plus's default, set the **LONG** option to a large value:

```
SQL> SELECT description, trigger_body
  2    FROM user_triggers
  3   WHERE trigger_name = 'MINIMUM_HOURLY_WAGE'
  4  /
...
SQL> SET LONG 4000
SQL> /
```

Try It:

Run *gettrig.sql* to dynamically generate a script which will create a trigger.

Enter **minimum_hourly_wage** as the trigger name.

This will create a *minimum_hourly_wage.sql* file!

LABS

❶ Give the manager of store #4 a 10% raise. Observe the results in both the **employee** table and the **pay_change_log** table. Make sure you have created the **pay_change_log** table and the **pay_change** trigger from this chapter.
(Solution: *mgr4raise.sql, paylog.sql*)

❷ Work with a partner:

a. Grant your partner **SELECT** and **UPDATE** privilege on your **employee** table. Your partner should do the same for you.

b. Update your partner's **employee** table, giving everyone at store #7 a 5% raise.

c. Observe the results in both of your **employee** tables and in your own **pay_change_log** table. Can you see your partner's changes to your employees? If not, why not? (Can you see your partner's **pay_change_log** table? If not, why not?)

(Solutions: *payloga.sql, paylogb.sql, paylogc.sql*)

❸ Make sure you have created the **minimum_hourly_wage** trigger from this chapter. Also, find a valid person **id** that isn't already in the **employee** table. Then, before attempting each of these, first predict what *should* happen. Compare with the actual results. Attempt to insert a new hourly employee (with a **pay_type_code** of **'H'**), using:

- A valid person **id**, with an invalid pay amount of $4.50.
- An invalid person **id**, with a valid pay amount of $6.50.
- An invalid person **id**, with an invalid pay amount of $4.50.

(Solution: *trigcheck.sql*)

❹ Create an **INSTEAD OF** trigger that will update both an employee and person record through a view. Create the following **VIEW**:

```
CREATE VIEW emp_view AS
SELECT id, lastname, firstname, title
  FROM employee JOIN person USING(id);
```

Write an **INSTEAD OF** trigger on this view that will update the correct table if an employee's name or title are updated.
(Solution: *emp_view_update.sql*)

Continued...

 © 2011 ITCourseware, LLC

PL/SQL compiler errors can be confusing.

For example, if there's an error in the trigger specification (that is, everything from **CREATE TRIGGER** up to the **DECLARE** line), then line numbers in compiler error messages start with the very first line of your specification.

However, if the specification is valid, but the trigger is created with compiler errors (because of syntax or other errors in the trigger body) then line numbers in the compiler error messages are counted starting at the first line of the trigger body (that is, the **DECLARE** line), not the first line of the specification.

LABS (CONT'D)

❺ Create a trigger to mark the logon time of a user, and another trigger for the logoff time. The triggers should insert a record into a table that stores the user name, time, and action. Test your trigger by logging in and out.
(Solutions: *logon_logoff.sql*, *check_logins.sql*)

❻ (Optional) Modify the minimum_hourly_wage trigger so that:

- It fires for both Hourly and Temporary employees.
- It selects the appropriate minimum wage, based on the pay type of the incoming record.

(Solution: *min_hourly_wage2.sql*)

 © 2011 ITCourseware, LLC

 © 2011 ITCourseware, LLC

CHAPTER 9 - USING PACKAGES

OBJECTIVES

- Invoke package procedures in SQL statements and PL/SQL programs.
- Use the **DBMS_OUTPUT** package to help debug PL/SQL programs.
- Use procedures in the **DBMS_UTILITY** package.
- Use packages to generate and parse XML data.
- Explore packages that trace and profile PL/SQL procedures and functions.
- Explore and use other Oracle Supplied Packages.

PACKAGES

- A *package* groups together a set of related PL/SQL objects, including procedures, functions, variables, datatypes, and cursors.

- Each package has a name, which must be distinct within its schema.

  - That is, if you have a package called **SHIPPING**, you can have only one such package within your schema.

- You reference package elements using dot-notation:

```
package_name.package_element_name;
```

- Each user session that accesses a package has its own copy of the variables declared in the package.

- Packages allow you to:

  - Organize stored subprograms into functional groups.

  - Save state information (through package variables) during a user's session.

  - Access (through Oracle's built-in packages) sophisticated capabilities of the Oracle server.

- Before you can use a package, its owner must grant you **EXECUTE** privileges on the package.

 © 2011 ITCourseware, LLC

Oracle-Supplied Packages

- Oracle provides a lot of non-SQL functionality in packages.

- You can use all of the supplied packages' procedures and functions from PL/SQL, and some of them from SQL statements.

  - **DBMS_OUTPUT** — Perform text output from PL/SQL programs
  - **DBMS_SQL** — Execute dynamic SQL, including DDL
  - **DBMS_JOB** — Schedule jobs within Oracle
  - **UTL_FILE** — Perform file I/O from PL/SQL
  - **DBMS_UTILITY** — Various utility routines
  - **DBMS_PIPE** — Communicate between Oracle sessions
  - **DBMS_STANDARD** — Trigger operations, transaction commands
  - **DBMS_SESSION** — Query and alter session information

- You can see the specification of a supplied package (its list of public variables and subprograms) in the **ALL_SOURCE** table.

```
SELECT text FROM all_source
 WHERE name = 'DBMS_UTILITY'
```

  - You can also use the SQL*Plus **DESCRIBE** command to get a short listing.

```
SQL> DESC DBMS_UTILITY
```

  - If the DBA has not granted you **EXECUTE** privilege on the package, you won't be able to see it.

```
SYSTEM> GRANT EXECUTE ON dbms_pipe TO dobbs;
```

 © 2011 ITCourseware, LLC

Some of the more userful supplied packages include:

DBMS_APPLICATION_INFO	Register applications with server for DBA monitoring
DBMS_DATA_MINING	Discover hidden patterns for use in making predictions
DBMS_DDL	Access to certain DDL activities from PL/SQL
DBMS_DESCRIBE	Describe arguments of a stored procedure
DBMS_EXPORT_EXTENSION	PL/SQL extensions to the export command
DBMS_JAVA	Access for Java-to-database functionality
DBMS_JOB	Job scheduling withing the Oracle server
DBMS_LOB	Large Object (**BLOB**, **CLOB**, **BFILE**) manipulation
DBMS_OUTPUT	Terminal output from PL/SQL programs
DBMS_RANDOM	Built-in random number generator
DBMS_REFRESH	Create refresh groups for replication snapshots
DBMS_REPAIR	Provides procedures for repair of corrupted data
DBMS_REPUTIL	Turn replication on and off
DBMS_ROWID	Manipulate restricted (v7) and extended (v8) **ROWID**s
DBMS_SESSION	Session manipulation from within PL/SQL
DBMS_SPACE	Analyze storage information
DBMS_SQL	Execute dynamic SQL, including DDL, in PL/SQL programs
DBMS_SQLTUNE	Tuning SQL statements
DBMS_TRANSACTION	Transaction manipulation from stored procedures
DBMS_UTILITY	Miscellaneous general-purpose procedures
UTL_COMPRESS	Collection of data compression utilities
UTL_FILE	Access operating system files from PL/SQL
UTL_HTTP	Request pages from Web servers from PL/SQL
UTL_INADDR	Support for internet addressing
UTL_RAW	Direct manipulation of **RAW** data

The DBMS_OUTPUT Package

- PL/SQL provides no native runtime I/O facilities.
- To observe the execution of your PL/SQL program, you can use the package **DBMS_OUTPUT**.
- The package allows you to define an output buffer to which your PL/SQL program can write.
  - Data in the buffer consists of lines, each up to 255 characters in length.
  - The default buffer size is 20,000 bytes, unless you use **ENABLE** to change it (up to 1,000,000 bytes).

```
DBMS_OUTPUT.ENABLE(300000);
```

- Several package procedures allow you to place data into the buffer.
  - **PUT** puts a string value into the buffer.
  - **PUT_LINE** puts a string value into the buffer, followed by a newline character.
  - **NEW_LINE** puts a newline character into the buffer.
- An application which runs your PL/SQL program can retrieve data from the buffer with the **GET_LINE** or **GET_LINES** procedures.
  - Lines retrieved are removed from the buffer automatically.
- The **DBMS_OUTPUT** package is intended primarily for debugging.
  - You cannot count on a program such as SQL*Plus to display this output from stored subprograms.

 © 2011 ITCourseware, LLC

If you are running your program from SQL*Plus, use **SET SERVEROUTPUT ON** to enable output.

```
SQL> SET SERVEROUTPUT ON
```

When the program finishes, SQL*Plus uses **DBMS_OUTPUT.GET_LINES** internally to get all lines from the buffer and print them to your terminal screen.

A simplified output procedure:

printproc.sql

```
CREATE OR REPLACE PROCEDURE print (msg VARCHAR2)
AS
BEGIN
   DBMS_OUTPUT.PUT_LINE(msg);
END;
/
```

This **print()** stored procedure makes it very easy to put data into the buffer.

```
SQL> BEGIN
   2    print ('DEBUG: Start of program');
   3    print ('DEBUG: Bottom of block');
   4  END;
   5 /

DEBUG: Start of program
DEBUG: Bottom of block
```

The DBMS_UTILITY Package

- The **DBMS_UTILITY** package contains a number of useful subprograms.
  - **GET_TIME** returns time in 100ths of a second; this is useful for customized benchmarking and debugging.
  - **COMPILE_SCHEMA** runs **ALTER OBJECT COMPILE** on all of your stored procedures, functions, packages, and triggers.
  - **ANALYZE_SCHEMA** runs **ANALYZE OBJECT ...** on all of your tables, clusters, and indexes.
  - **FORMAT_ERROR_STACK** gathers information about an error's propagation into a string variable; useful in exception handlers.
  - **GET_PARAMETER_VALUE** returns the current value of an Oracle initialization parameter (from *init.ora*).
  - **NAME_TOKENIZE**, **COMMA_TO_TABLE**, and **TABLE_TO_COMMA** perform specialized string parsing.
  - . . . And so on; there are several others.

 © 2011 ITCourseware, LLC

THE UTL_FILE PACKAGE

- To access files from a PL/SQL program, use the **UTL_FILE** package.

- The steps for performing file I/O with **UTL_FILE** resemble those of other languages.

  1. Declare a file handle variable (and variables for data).

```
fh UTL_FILE.FILE_TYPE;
line VARCHAR2(500);
```

  2. Open the file, specifying its name, file path (8*i*) or directory object (9*i*), and mode (**r** for reading, **w** for writing).

```
fh := UTL_FILE.FOPEN('/app/files','f1.txt','r'); -- 8i
fh := UTL_FILE.FOPEN('APPFILES','f1.txt','r'); -- 9i
```

  3. Perform your I/O.

```
UTL_FILE.GET_LINE(fh, line);
UTL_FILE.PUT_LINE(fh, line);
```

  4. Close the file, flushing written data and freeing resources.

```
UTL_FILE.FCLOSE(fh);
```

- Other **UTL_FILE** methods include:

  - **PUT(fh, s)** puts a string in the file, without a line terminator.

  - **PUTF(fh, s, ...)** puts a formatted string in the file, with string substitution for **%s** and newline for **\n** (like a limited C language **printf()**).

  - **FFLUSH(fh)** flushes all output to the file.

 © 2011 ITCourseware, LLC

Prior to Oracle9*i:*

For you to use **UTL_FILE**, the DBA must specify locations in which Oracle users can access files. By default, no operating system files can be accessed from within Oracle. To specify a location, the DBA must add a directive to the Oracle server initialization file (i.e., *init.ora*):

```
UTL_FILE_DIR = /app/files
```

FOPEN will throw an **INVALID_PATH** exception if you try to open a file anywhere else. The DBA can specify as many locations as necessary:

```
UTL_FILE_DIR = /app/files/logs
UTL_FILE_DIR = /home/orausers/tmp
```

Using ***** as the location name allows file access throughout the host operating system:

```
UTL_FILE_DIR = *
```

Oracle9*i* and later:

In Oracle9*i*, the **UTL_FILE_DIR** directive was replaced with **DIRECTORY** objects. Directory objects are associated with specific directories on the file system. Only users with the **CREATE ANY DIRECTORY** system privilege are allowed to create directory objects.

```
CREATE OR REPLACE DIRECTORY directory_object_name
 AS 'system path';
```

Only directories associated with directory objects can be accessed by a PL/SQL program. Oracle also must have operating system privileges to use the specified directory.

```
CREATE OR REPLACE DIRECTORY appfiles
 AS '/app/files';
```

The DBMS_METADATA Package

- The **DBMS_METADATA** package provides an API to extract DDL and XML from the Data Dictionary for database objects.
  - You may also transform metadata in XML format to DDL or re-create database objects from XML.
- The **GET_XML** and **GET_DDL** functions retrieve object metadata as a **CLOB** in a single call, with parameters for object type, name, and schema (which defaults to the current user's schema):

```
SELECT DBMS_METADATA.GET_DDL('TABLE', 'EMPLOYEE', 'S1') FROM dual;
SELECT DBMS_METADATA.GET_XML('TRIGGER','EMP_TRIG','S1') FROM dual;
```

- **GET_DEPENDENT_DDL** and **GET_DEPENDENT_XML** retrieve an object's dependent metadata, such as indexes and foreign keys:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL('INDEX','PERSON','S1') FROM dual;
```

- **GET_GRANTED_DDL** and **GET_GRANTED_XML** extract object, system, and role grants given to a particular schema:

```
SELECT DBMS_METADATA.GET_GRANTED_XML('SYSTEM_GRANT','S1') FROM dual;
```

- You can retrieve metadata on heterogeneous objects, such as complete schemas, by using **OPEN**, **SET_FILTER**, **FETCH_xxx**, and **CLOSE**.
  - **OPEN** — opens a numeric handle for the set of queried objects.
  - **CLOSE** — closes the handle
  - **SET_FILTER** — specifies which object(s) to extract; you may set more than one filter for a call
  - **FETCH_CLOB**, **FETCH_XML**, and **FETCH_DDL** — return object metadata in **CLOB**, **SYS.XMLTYPE**, and **SYS.KU$_DDL** format, respectively

 © 2011 ITCourseware, LLC

The following example creates a function to extract XML metadata about a specified table.

xmlmeta.sql

```
CREATE OR REPLACE FUNCTION get_xml_meta(tablename VARCHAR2)
  RETURN clob
IS
   handle NUMBER;
   doc CLOB;
BEGIN
   handle := DBMS_METADATA.OPEN('TABLE');
   DBMS_METADATA.SET_FILTER(handle,'SCHEMA',USER);
   DBMS_METADATA.SET_FILTER(handle,'NAME',tablename);
   doc := DBMS_METADATA.FETCH_CLOB(handle);
   DBMS_METADATA.CLOSE(handle);
   RETURN doc;
END;
/
```

This example uses a transformer to transform the XML metadata into a SQL DDL statement.

ddlmeta.sql

```
CREATE OR REPLACE FUNCTION get_ddl_meta(tablename VARCHAR2)
 RETURN CLOB
IS
     handle NUMBER;
     temp_handle NUMBER;
     doc CLOB;
BEGIN
     handle := DBMS_METADATA.OPEN('TABLE');
     DBMS_METADATA.SET_FILTER(handle,'SCHEMA',USER);
     DBMS_METADATA.SET_FILTER(handle,'NAME',tablename);
     temp_handle := DBMS_METADATA.ADD_TRANSFORM(handle,'DDL');
     doc := DBMS_METADATA.FETCH_CLOB(handle);
     DBMS_METADATA.CLOSE(handle);
     RETURN doc;
END;
/

SELECT get_ddl_meta('EMPLOYEE') FROM dual;
```

The **KU$_DDL** datatype consists of the DDL text and a table of parsed items. You can describe the datatype in SQL*Plus:

```
SQL> SET DESCRIBE DEPTH ALL
SQL> DESC KU$_DDL
```

XML Packages

- Introduced in 9*i*, you can modify the output of a SQL query into XML format using **DBMS_XMLGEN**.
  - The **GETXML** function returns an arbitrary SQL **SELECT** statement's output in XML format.

```
SELECT DBMS_XMLGEN.GETXML('SELECT * FROM store') FROM dual;
```

  - **DBMS_XMLGEN** allows you to specify the number of rows to return and whether rows should be skipped.

- Use the **DBMS_XMLPARSER** package to parse the content and organization of XML files.
  - The Oracle PL/SQL XML parser will build a parse tree structure that is accessible by DOM APIs.
  - The parser can validate the XML against a DTD (Document Type Definition), but not a schema.

- Released with 10*g*, **DBMS_XMLSTORE** provides an API to store XML data in tables.
  - Use **SETUPDATECOLUMN** to specify which columns will be inserted or updated.
    - All other columns will receive null or default values on an **INSERT**, or will be left unchanged on an **UPDATE**.
  - The **INSERTXML** function will use the XML data and insert new rows into the context table.
  - The **UPDATEXML** and **DELETEXML** functions use the key defined in **SETKEYCOLUMN** to determine which rows to update or delete.

 © 2011 ITCourseware, LLC

The following example uses **DBMS_XMLSTORE** to insert new entries into the **employee** table using the information stored in an XML document. Two new rows will be inserted into the **s1** schema's **employee** table.

xmlinsert.sql

```
SET SERVEROUTPUT ON
DECLARE
   context DBMS_XMLSTORE.ctxType;
   rowcount NUMBER;
   xmlData CLOB :=      --  XML data set here
      '<ROWSET>
         <ROW num="1">
            <ID>1</ID>
            <FIRSTNAME>Ted</FIRSTNAME>
            <LASTNAME>Striker</LASTNAME>
         </ROW>
      <ROW>
         <ID>2</ID>
         <FIRSTNAME>Inigo</FIRSTNAME>
         <LASTNAME>Montoya</LASTNAME>
      </ROW>
   </ROWSET>';
BEGIN
   context := DBMS_XMLSTORE.NEWCONTEXT('S1.PERSON'); -- get context
   DBMS_XMLSTORE.CLEARUPDATECOLUMNList(context); -- clear update settings
   DBMS_XMLSTORE.SETUPDATECOLUMN(context,'ID');  -- set fields to be updated
   DBMS_XMLSTORE.SETUPDATECOLUMN(context,'FIRSTNAME');
   DBMS_XMLSTORE.SETUPDATECOLUMN(context,'LASTNAME');
   rowcount := DBMS_XMLSTORE.INSERTXML(context, xmlData); -- insert new records
   DBMS_XMLSTORE.CLOSECONTEXT(context);
   DBMS_OUTPUT.PUT_LINE(rowcount || ' new rows inserted in person table.');
END;
/
```

The following example will delete rows in the **person** table with id 1 and 2.

xmldelete.sql

```
SET SERVEROUTPUT ON
DECLARE
   context DBMS_XMLSTORE.CtxType;
   rowcount NUMBER;
   xmlData CLOB :=      --  XML data set here
      '<ROWSET>
         <ROW num="1">
            <ID>1</ID>
         </ROW>
      <ROW>
         <ID>2</ID>
      </ROW>
   </ROWSET>';
BEGIN
   context := DBMS_XMLSTORE.NEWCONTEXT('S1.PERSON'); -- get context
   DBMS_XMLSTORE.SETKEYCOLUMN(context,'ID');
   rowcount := DBMS_XMLSTORE.DELETEXML(context, xmlData);
   DBMS_XMLSTORE.CLOSECONTEXT(context);
   DBMS_OUTPUT.PUT_LINE(rowcount || ' rows where deleted');
END;
/
```

NETWORKING PACKAGES

- The **UTL_HTTP** package lets you access data over the internet using protocols such as HTTP, HTTPS, FTP, and Gopher.
  - You can retrieve the first 2000 bytes at a URL from a PL/SQL block or SQL statement:

```
SELECT UTL_HTTP.REQUEST('http://www.example.com') FROM dual;
```

- For reading web pages longer than 2000 bytes from a PL/SQL block, you must use several steps:

  1. Establish a persistent network connection (**BEGIN_REQUEST**)
  2. Set any optional headers (**SET_HEADER**)
  3. Read the response (**GET_RESPONSE**)
  4. Loop through the response lines (**READ_LINE**)
  5. Close the connection (**END_RESPONSE**)

  - **SET_PROXY** allows you to configure proxy settings for a session.

- 10*g* introduced the **UTL_MAIL** package to send email.
  - The **SEND** procedure packages and sends the email using the SMTP server; you must specify the sender and comma separated list of recipients; the cc, bcc, subject, message, mime type, and priority parameters are all optional.
  - **SEND_ATTACH_RAW** and **SEND_ATTACH_VARCHAR2** allow for binary and character attachments, respectively.

- **DBMS_LDAP** provides functions and procedures to access data from LDAP.
  - **LDAP** is an extensible directory protocol.
  - It is a common language that clients and servers use to communicate.

 © 2011 ITCourseware, LLC

The following example will open a request to an HTTP server and read the response lines. When **READ_LINE** reads past the last response line it throws an **END_OF_BODY** exception.

http.sql

```
SET SERVEROUTPUT ON
DECLARE
   line      VARCHAR2(2048);
   request   UTL_HTTP.req;
   response  UTL_HTTP.resp;
BEGIN
   request := UTL_HTTP.BEGIN_REQUEST('http://www.example.com');
   UTL_HTTP.SET_HEADER(request, 'User-Agent', 'Mozilla/4.0');
   response := UTL_HTTP.GET_RESPONSE(request);
   LOOP
      UTL_HTTP.READ_LINE(response, line, TRUE);
      DBMS_OUTPUT.PUT_LINE(substr(line,1,255));
   END LOOP;
   UTL_HTTP.END_RESPONSE(response);
EXCEPTION
   WHEN UTL_HTTP.END_OF_BODY THEN
      UTL_HTTP.END_RESPONSE(response);
END;
/
```

mail.sql

```
BEGIN
   UTL_MAIL.SEND(sender => 's2@domain.com',
                 recipients => 's1@domain.com',
                 subject    => 'Testing UTL_MAIL',
                 message    => 'This is a test of UTL_MAIL.SEND.'
                               'This is only a test.');
END;
/
```

UTL_MAIL:

For **UTL_MAIL** to work, two setup operations must be performed.

1. The DBA must install **UTL_MAIL**:
   $ORACLE_HOME/rdbms/admin/utlmail.sql
   $ORACLE_HOME/rdbms/admin/prvtmail.plb

2. The **SMTP_OUT_SERVER** parameter should be defined in the database initialization file (defaults to **DB_DOMAIN** initilization parameter).

OTHER SUPPLIED PACKAGES

- You can trace PL/SQL code with **DBMS_TRACE**.
  - The trace data is collected in previously-defined database tables as the program executes.
  - A typical session involves:
    1. Calling **SET_PLSQL_TRACE** to start the tracing.
       - **SET_PLSQL_TRACE** can set tracing for various levels including **TRACE_ALL_CALLS**, **TRACE_ALL_EXCEPTIONS**, **TRACE_ALL_SQL**, and **TRACE_ALL_LINES**.
    2. Running the application.
    3. Calling **CLEAR_PLSQL_TRACE** to stop the tracing.
- Updated in 10*g*, **DBMS_PROFILER** collects performance information on existing PL/SQL code to determine performance bottlenecks; performance information includes:
  - Line execution count
  - Time spent executing each line
  - Min and max time needed to execute a particular line
  - As with **DBMS_TRACE**, the profiling information is stored in database tables which must be queried for results.
- Introduced in 10*g*, **DBMS_CRYPTO** provides for encrypting and decrypting **RAW**, **BLOB**, and **CLOB** datatypes.
  - **ENCRYPT** will encrypt data using a stream or block cipher with a user-supplied key and optional initialization vector.
  - Use **DECRYPT** with the same cipher, etc. to get original data.

 © 2011 ITCourseware, LLC

The following example calls a function that returns the current number of employees in the **employee** table. The function is called three times, with different types of tracing applied, and then a fourth time to collect profile data.

trace.sql

```
DECLARE
   c NUMBER;
   result BINARY_INTEGER;
   FUNCTION countEmps RETURN NUMBER
   IS
      empCount NUMBER;
   BEGIN
      SELECT count(*)
         INTO empCount
         FROM employee;
      RETURN empCount;
   END;
BEGIN
   DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_CALLS);
   c := countEmps;
   DBMS_TRACE.CLEAR_PLSQL_TRACE;
   DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_SQL);
   c := countEmps;
   DBMS_TRACE.CLEAR_PLSQL_TRACE;
   DBMS_TRACE.SET_PLSQL_TRACE (DBMS_TRACE.TRACE_ALL_LINES);
   c := countEmps;
   DBMS_TRACE.CLEAR_PLSQL_TRACE;
   result := DBMS_PROFILER.START_PROFILER(run_comment => 'countEmps - ' ||
           SYSDATE);
   c := countEmps;
   result := DBMS_PROFILER.STOP_PROFILER;
END;
/
```

After the above code is run, the following query can be used to acquire the **runid** for a particular trace. A similar query can be run on the **PLSQL_PROFILER_RUNS** table to retrieve the profile **runid**.

trace_runid.sql

```
   SELECT runid, run_owner,
           TO_CHAR(run_date, 'MON DD, YYYY HH:MI:SS AM') AS run_date
     FROM plsql_trace_runs
ORDER BY runid;
```

You then use the **runid** to retrieve the trace information. The following example will retrieve trace information using event **runid 12**. A similar query can be run on the **PLSQL_PROFILER_UNITS** and **PLSQL_PROFILER_DATA** tables to retrieve profile information.

trace_data.sql

```
   SELECT runid, event_seq,
           TO_CHAR(event_time, 'MON DD, YYYY HH:MI:SS AM') AS time,
            event_unit_owner, event_unit, event_unit_kind, proc_line,
          event_comment
     FROM plsql_trace_events
    WHERE runid = 12
ORDER BY runid, event_seq;
/
```

LABS

❶ Write a program that uses **DBMS_OUTPUT** to print "Hello, World!"
(Solution: *hello.sql*)

❷ Write a **SELECT** statement to read the first ten lines returned by *www.w3c.org*.
(Solution: *w3c.sql*)

❸ Write a **SELECT** statement that will return the firstname, lastname, and city of all persons in Colorado and return the information in XML format.
(Solution: *coloradoxml.sql*)

❹ Insert yourself into the **person** table using XML data.
(Solution: *selfinsert.sql*)

 © 2011 ITCourseware, LLC

 © 2011 ITCourseware, LLC

CHAPTER 18 - DEBUGGING AND ERROR HANDLING

OBJECTIVES

- Handle PL/SQL program exceptions.
- Trace PL/SQL program execution for debugging purposes.
- Use Oracle SQL Developer to aide in developing and debugging PL/SQL.
- Learn about commonly found bugs and how to avoid them.

EXCEPTION MANAGEMENT

- PL/SQL blocks can have two different types of errors: compile-time errors and runtime errors.
  - *Compile-time errors* stop the PL/SQL block from compiling successfully.
    - Examples of compile time errors include invalid syntax, insufficient permissions, and referencing missing objects.
  - *Runtime errors* occur when the PL/SQL block encounters a problem while executing.
    - Examples of runtime errors include dividing by zero, type mismatches, and no data found for **SELECT ... INTO** statements.
- *Exceptions* are routines in your PL/SQL code to handle a variety of runtime errors.
  - When a runtime error occurs, the exception is raised and the code handles the exception.
- The exception handling section in the code comes last in the PL/SQL block:

```
DECLARE
   <declaration section>
BEGIN
   <code section>
EXCEPTION
   <exception section>
END;
/
```

- Once the exception code is executed, the PL/SQL block terminates.
  - Only the first exception matching the runtime error is executed.

 © 2011 ITCourseware, LLC

Exception handling will perform any actions you code in the event of a runtime error.

exception_example.sql

```
SET SERVEROUTPUT ON
DECLARE
  num_items NUMBER;
  total_price NUMBER;
  average NUMBER;
BEGIN
  SELECT SUM(quantity) INTO num_items FROM oe.order_items;
  SELECT SUM(unit_price) INTO total_price FROM oe.order_items;
  average := total_price / num_items;
  DBMS_OUTPUT.PUT_LINE('Average per item = ' || average);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('No items sold');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Order items is empty');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An internal error occurred.');
END;
/
```

With the code above, there are a few errors that could occur on runtime. The **SUM(*items*)** could result in zero. The computation of the average would attempt to divide by zero, which is an error. The **ZERO_DIVIDE** exception is thrown, displaying the appropriate message. Either of the **SELECT ... INTO** statements could result in the **NO_DATA_FOUND** exception.

The **WHEN OTHERS** section is a catchall exception. When a runtime error occurs that is not one of the two previous exceptions, the code in the **WHEN OTHERS** section is executed.

Only one **WHEN *exception_name* THEN** section will be executed in the PL/SQL block.

```
...
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('This exception can be raised');
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('You will never see me.');
END;
```

EXCEPTION PROPAGATION

- When an exception is raised, the local PL/SQL block is examined for the exception handler code.
- If the exception handler code is found in the local block, the exception is handled locally and the local block is terminated.
- If the exception handler code is not found in the local block, the exception is propagated or passed up to the calling block.
  - The parent block takes control of the exception handling.
- If no exception handler code is found in the local block, or any of its predecessors, an unhandled exception error is returned to the host.

 © 2011 ITCourseware, LLC

propagation_example.sql

```
DECLARE
  val NUMBER;
BEGIN
  SELECT salary INTO val FROM hr.employees
   WHERE employee_id = 206;

  -- Subprogram begins here
  BEGIN
    SELECT salary INTO val FROM hr.employees
     WHERE employee_id = 207;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No salary for employee 207');
  END;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No salary for employee 206');
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('An employee has two salaries');
END;
/
```

If there are no employees with id **207**, then the **NO_DATA_FOUND** exception in the sub-block is raised. Since there is a **NO_DATA_FOUND** exception handler local to the sub-block, the host will receive a message indicating that there is no salary for employee number **207**.

The **SELECT ... INTO** statement will only work if only one row of data is returned. If more than one row is returned, the **TOO_MANY_ROWS** exception is raised. If the sub-block finds more than one employee with id **207**, then it will raise the **TOO_MANY_ROWS** exception. There is no exception handler for this error local to the sub-block, so the exception is propagated to the calling block. The host will receive a message indicating that there are two salaries for that employee.

In the main section of code, it is possible for the query on employee **206** to return no rows of data. In this case, the exception local to that section of code is executed. The host will receive a message indicating that there is no salary for employee number **206**.

User-Defined Exceptions

- The **DBMS_STANDARD** package includes predefined exceptions available for use in any PL/SQL block.
  - Nothing special needs to be done to use the predefined exceptions.
- The application developer can create their own exceptions to extend what is already available.
  - The user-defined exception name must be declared along with any variables in the block's declaration section.

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF <condition> THEN
   RAISE exception_name;
END IF;
EXCEPTION
   WHEN exception_name THEN
      <handler code>
END;
```

  - You must explicitly raise the exception, since Oracle will not throw it for you.
    - Normally, a conditional determines if the exception should be raised.

 © 2011 ITCourseware, LLC

Predefined exceptions available through **DBMS_STANDARD**:

ACCESS_INTO_NULL	**LOGIN_DENIED**	**SUBSCRIPT_BEYOND_COUNT**
CASE_NOT_FOUND	**NO_DATA_FOUND**	**SUBSCRIPT_BEYOND_LIMITS**
COLLECTION_IS_FULL	**NOT_LOGGED_IN**	**TIMEOUT_ON_RESOURCE**
CURSOR_ALREADY_OPEN	**PROGRAM_ERROR**	**TOO_MANY_ROWS**
DUP_VALUE_ON_INDEX	**ROWTYPE_MISMATCH**	**VALUE_ERROR**
INVALID_CURSOR	**SELF_IS_NULL**	**ZERO_DIVIDE**
INVALID_NUMBER	**STORAGE_ERROR**	

check_limit.sql

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE check_limit(
  cust_id IN NUMBER, oid NUMBER)
AS
   insufficient_terms EXCEPTION;
  limit NUMBER;
  total NUMBER;
BEGIN
  SELECT credit_limit INTO limit
    FROM oe.customers WHERE customer_id = cust_id;
  SELECT order_total INTO total
    FROM oe.orders WHERE order_id = oid;

  IF (total > limit) THEN
    RAISE insufficient_terms;
  END IF;
  UPDATE oe.orders SET order_status = 1
   WHERE order_id = oid;
EXCEPTION
  WHEN insufficient_terms THEN
    UPDATE oe.orders SET order_status = -1
     WHERE order_id = oid;
    DBMS_OUTPUT.PUT_LINE('Order ($' || total ||
      ') exceeds credit limit ($' || limit || ')');
END;
/
```

Try It:

Run *check_these.sql* to test this procedure.

EXCEPTION ERROR MESSAGES

- Exceptions can be traced to a specific Oracle error message (**ORA-*xxxxx***).
  - The **SQLCODE** built-in function can be used to obtain the error number.
  - The **SQLERRM** built-in function returns the error message.
- You can create your own error code messages with the **RAISE_APPLICATION_ERROR** procedure.

```
RAISE_APPLICATION_ERROR(err_num, message[, (TRUE|FALSE)] );
```

  - The error code must be a negative number between -20000 and -20999.
    - All other error codes are reserved for Oracle's use.
  - The error message is limited to 2048 characters.
  - The last parameter is optional and determines how the error is placed on the error stack.
    - **TRUE** — add error to the top of the error stack.
    - **FALSE** — replace all previous errors on the stack with this error (this is the default).

 © 2011 ITCourseware, LLC

We have seen how the **WHEN OTHERS** clause is a catchall exception handler. The following code sample shows how to return more information, such the Oracle error code and message that threw the exception in the **WHEN OTHERS** section.

err_codes.sql

```
DECLARE
  qty NUMBER;
  err_code NUMBER;
  err_msg VARCHAR2(100);
BEGIN
  SELECT quantity_on_hand INTO qty
    FROM inventories
   WHERE product_id = 999;
EXCEPTION
  WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('An exception occurred.');
    err_code := SQLCODE;
    err_msg := SUBSTR(SQLERRM, 1, 100);
    DBMS_OUTPUT.PUT_LINE('Error code: ' || err_code);
    DBMS_OUTPUT.PUT_LINE('Error msg: ' || err_msg);
END;
/
```

raise_error.sql

```
DECLARE
  qty NUMBER;
BEGIN
  SELECT quantity_on_hand INTO qty
    FROM oe.inventories
   WHERE product_id = 2870;

  IF (qty < 100) THEN
     RAISE_APPLICATION_ERROR(-20001, 'Need to order more');
  END IF;

END;
/
```

Stack Management

- It may be useful to determine the order of calls to various PL/SQL procedures and functions.
  - The **DBMS_UTILITY.FORMAT_CALL_STACK** function will return the call stack, up to 2000 characters.
- You may also find it useful to see the error stack with all of the error messages.
  - The **DBMS_UTILITY.FORMAT_ERROR_STACK** function returns the error stack, up to 2000 characters.
- When debugging PL/SQL code, it is very useful to show the call stack at the point of the exception.
  - The **DBMS_UTILITY.FORMAT_ERROR_BACKTRACE** function returns the call stack at the exception, up to 2000 characters.

 © 2011 ITCourseware, LLC

The function **test_fn** returns the square root of a number. The **test_proc** procedure calls this function. If an exception is raised, we'll get the error stack and the error backtrace.

test_fn.sql

```
CREATE OR REPLACE FUNCTION test_fn (x NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN SQRT(x);
END;
/
```

test_proc.sql

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE test_proc
IS
  val NUMBER;
  msg VARCHAR2(2000);
BEGIN
  SELECT test_fn(-1) INTO val FROM dual;
EXCEPTION
  WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('Error Stack...');
     msg := DBMS_UTILITY.FORMAT_ERROR_STACK;
     DBMS_OUTPUT.PUT_LINE(msg);
     DBMS_OUTPUT.PUT_LINE('Error Backtrace...');
     msg := DBMS_UTILITY.FORMAT_ERROR_BACKTRACE;
     DBMS_OUTPUT.PUT_LINE(msg);
END;
/
```

Execute the stored procedure.

```
SQL> EXEC test_proc;
Error Stack...
ORA-06502: PL/SQL: numeric or value error

Error Backtrace...
ORA-06512: at "HR.TEST_FN", line 4
ORA-06512: at "HR.TEST_PROC", line 6

PL/SQL procedure successfully completed.
```

The backtrace shows that the **test_fn** function had the error and it was called by the **test_proc** procedure. Oracle uses these **DBMS_UTILITY** functions when executing stored procedures in SQL*Plus, so the messages should be familiar.

Debugging with DBMS_OUTPUT

- The **DBMS_OUTPUT** supplied package is one tool for debugging PL/SQL programs.
- Place **DBMS_OUTPUT.PUT_LINE** statements throughout the PL/SQL block to show progress within the block.
  - Use the **PUT** or **PUT_LINE** procedure to write contents of variables and watch them change while the program runs.

```
DBMS_OUTPUT.PUT_LINE('After loop, i=' || i);
```

- All database users have access to the **DBMS_OUTPUT** package.
  - You will need to issue **SET SERVEROUTPUT ON** in SQL*Plus to see **DBMS_OUTPUT** lines.
- Output from **DBMS_OUTPUT** is limited to 32,767 characters per line.
- Output from the **DBMS_OUTPUT** procedures is not seen until the buffer is flushed.
  - Typically, the buffer is not flushed until the PL/SQL block stops execution.

 © 2011 ITCourseware, LLC

Many application developers are used to using statements which output debug information as the program runs. In Java, one might use the **System.out.println** procedure. In C++, you might use **printf** or **cout**. The equivalent in PL/SQL is the **DBMS_OUTPUT.PUT_LINE** procedure. Notice in the output from the following code sample how you can tell where in the code you are. The following example also shows the contents of the variable changing inside the loop.

show_output.sql

```
SET SERVEROUTPUT ON
DECLARE
  i NUMBER;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Start of PL/SQL block');
  FOR i IN 1..5 LOOP
      DBMS_OUTPUT.PUT_LINE('i=' || i);
  END LOOP;
   DBMS_OUTPUT.PUT_LINE('Loop done processing');
  DBMS_OUTPUT.PUT_LINE('After loop, i=' || i);
   DBMS_OUTPUT.PUT_LINE('End of PL/SQL block');
END;
/
```

```
Start of PL/SQL block
i=1
i=2
i=3
i=4
i=5
Loop done processing
After loop, i=
End of PL/SQL block

PL/SQL procedure successfully completed.
```

Debugging with a Table

- When working with long-running PL/SQL blocks, it may take a while for the output from **DBMS_OUTPUT** to appear on your screen.
- Writing debug information to a table can be beneficial, since you can query the debug table in another session while the PL/SQL block is executing.
- Output inserted into a table can persist long after the PL/SQL block has been executed.
- To debug with a table, you will need a debug table and a stored procedure to write into that table.
  - The stored procedure is defined as an autonomous transaction: the messages do not get removed if the PL/SQL block rolls back any transactions.

 © 2011 ITCourseware, LLC

First, a debug table needs to be created. We'll also create a sequence to ensure ordering of our debug statements. Then, we'll create a stored procedure to insert debug information into this table.

setup_debug.sql

```
CREATE TABLE debug_table (
   id NUMBER,
   msg VARCHAR2(60)
);

CREATE SEQUENCE debug_seq START WITH 1 INCREMENT BY 1;

CREATE OR REPLACE PROCEDURE debug_msg (msg IN VARCHAR2) AS
    PRAGMA AUTONOMOUS_TRANSACTION;
   seq_num NUMBER;
BEGIN
    SELECT debug_seq.NEXTVAL INTO seq_num FROM dual;
    INSERT INTO debug_table VALUES (seq_num, msg);
   COMMIT;
END;
/
```

Now that the procedure is created, we can use it to debug a PL/SQL block.

show_debug.sql

```
BEGIN
  FOR i IN 1..5 LOOP
    debug_msg('i=' || i);
  END LOOP;
END;
/

SELECT * FROM debug_table;
```

```
        ID MSG
---------  ------------------------------------------------------------
         1 i=1
         2 i=2
         3 i=3
         4 i=4
         5 i=5
```

Using UTL_FILE

- The **UTL_FILE** supplied package lets a PL/SQL program read and write operating system files.
- The PL/SQL code can write debug information to the operating system file.
  - This is very useful when the volume of debug information would scroll past the screen too fast.
- Good applications write information to log files to be analyzed in the event of a failure.
  - The instance's Alert Log is an example of this type of log file.
  - The **UTL_FILE** package can be used to create your own application log.
- Many database operations are audited through Oracle's **AUDIT** command.
  - The **UTL_FILE** package can be used to implement auditing of the application to an audit trail.
- The **UTL_FILE** package interacts with operating system files found in the file system's directory pointed to by a **DIRECTORY** database object.
  - The DBA may need to create a **DIRECTORY** object and grant appropriate privileges to the user.

 © 2011 ITCourseware, LLC

setup_utl_file.sql

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

Now that the directory has been created, it is used with **UTL_FILE.**

utl_file_example.sql

```
DECLARE
   f1 UTL_FILE.FILE_TYPE;
  x NUMBER;
BEGIN
   f1 := UTL_FILE.FOPEN('TMP_DIRECTORY', 'app_log.txt', 'w');

   UTL_FILE.PUT_LINE(f1, TO_CHAR(sysdate, 'MM-DD-YYYY HH24:MI:SS') ||
     ' Entered PL/SQL block', TRUE);

  FOR x IN 1..5 LOOP
     UTL_FILE.PUT_LINE(f1, TO_CHAR(sysdate, 'MM-DD-YYYY HH24:MI:SS') ||
       ' Inside loop. x=' || x, TRUE);
  END LOOP;

   UTL_FILE.PUT_LINE(f1, TO_CHAR(sysdate, 'MM-DD-YYYY HH24:MI:SS') ||
     ' Leaving PL/SQL block', TRUE);

   UTL_FILE.FCLOSE(f1);
END;
/
```

The contents of the */tmp/app_log.txt* file can be seen below:

```
04-12-2006 19:53:23 Entered PL/SQL block
04-12-2006 19:53:23 Inside loop. x=1
04-12-2006 19:53:23 Inside loop. x=2
04-12-2006 19:53:23 Inside loop. x=3
04-12-2006 19:53:23 Inside loop. x=4
04-12-2006 19:53:23 Inside loop. x=5
04-12-2006 19:53:23 Leaving PL/SQL block
```

Using DBMS_DEBUG

- The **DBMS_DEBUG** supplied package is an interface to the Oracle PL/SQL debugger.
- With the **DBMS_DEBUG** package, you can perform a variety of operations to debug your PL/SQL code.
  - Set breakpoints to stop execution of the PL/SQL code at a specific line.
  - Remove breakpoints previously set with the code.
  - Obtain information about the execution of the code.
  - See variable contents.
  - Set a variable to a specific value any time during code execution.
  - Step in to called functions or procedures.
  - Step out of the called subprogram.

 © 2011 ITCourseware, LLC

The following table shows all of the procedures of the **DBMS_DEBUG** supplied package:

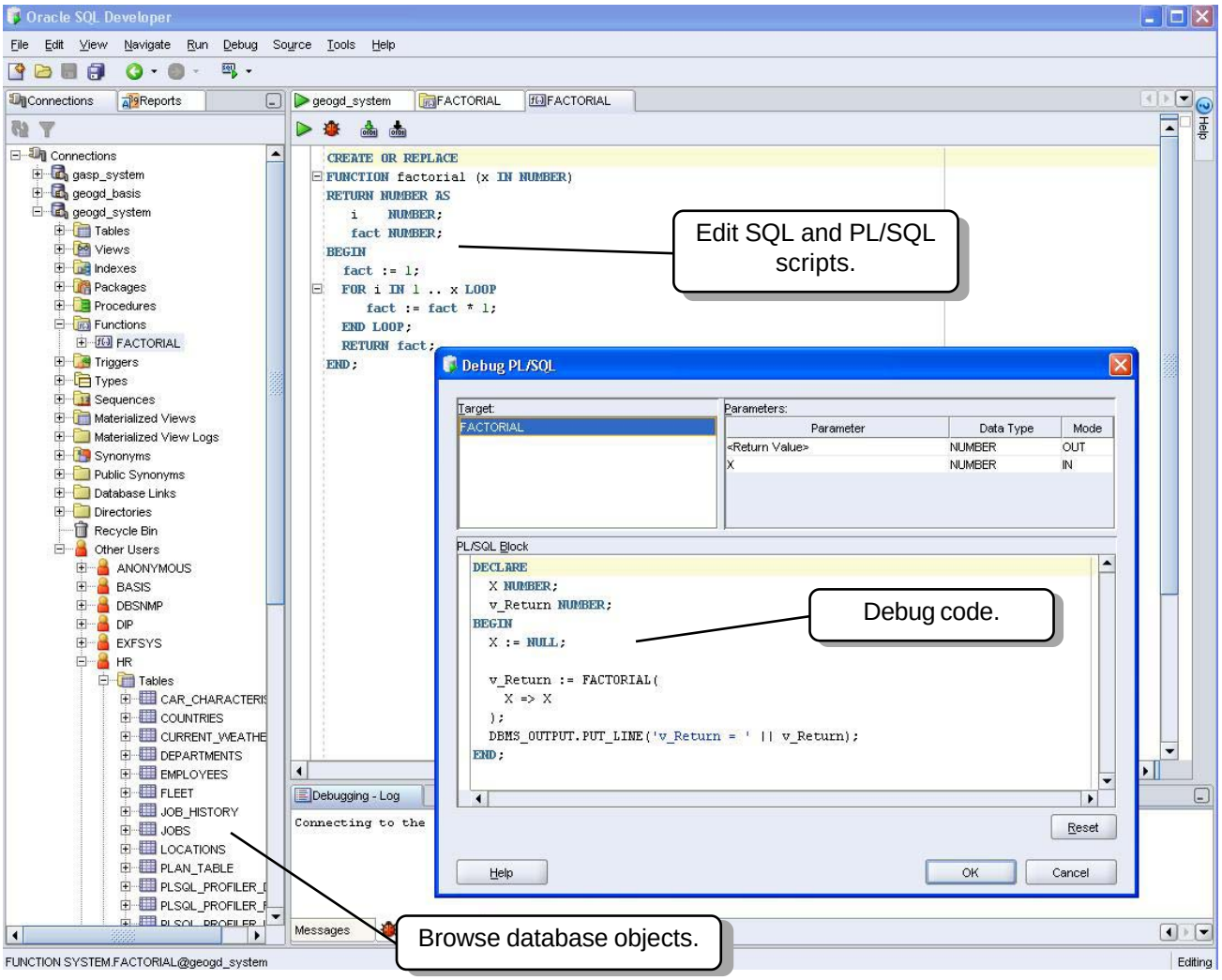
ATTACH_SESSION	**GET_MORE_SOURCE**	**SET_BREAKPOINT**
CONTINUE	**GET_LINE_MAP**	**SET_OER_BREAKPOINT**
DEBUG_OFF	**GET_RUNTIME_INFO**	**SET_TIMEOUT**
DEBUG_ON	**GET_TIMEOUT_BEHAVIOR**	**SET_TIMEOUT_BEHAVIOR**
DELETE_BREAKPOINT	**GET_VALUE**	**SET_VALUE**
DELETE_OER_BREAKPOINT	**INITIALIZE**	**SHOW_BREAKPOINTS**
DETACH_SESSION	**PING**	**SHOW_FRAME_SOURCE**
DISABLE_BREAKPOINT	**PRINT_BACKTRACE**	**SHOW_SOURCE**
ENABLE_BREAKPOINT	**PRINT_INSTANTIATIONS**	**SYNCHRONIZE**
EXECUTE	**PROBE_VERSION**	**TARGET_PROGRAM_RUNNING**
GET_INDEXES	**SELF_CHECK**	

SQL Developer

- Oracle SQL Developer is a GUI tool for SQL and PL/SQL development.
- This Java program can be obtained, for free, from Technet at ***http://www.oracle.com/technology/products/database/sql_developer/index.html***.
- SQL Developer is a great editor for working with PL/SQL code.
  - Use Code Insight for automatic code completion.
  - Use Code Formatter for easier reading of PL/SQL code.
- SQL Developer will quickly become a favorite tool for PL/SQL debugging.
  - Step into or step over called functions or procedures.
  - Set and manage breakpoints.
  - Inspect and modify variables.
- SQL Developer is much easier to use than **DBMS_DEBUG**.

 © 2011 ITCourseware, LLC

A few of the features of SQL Developer:



Avoiding Bugs

- Add error checking wherever you can to avoid exceptions.
  - Add exception handlers where errors can occur.
    - Use named exception handlers where possible and do not rely solely on the **WHEN OTHERS** exception handler.
  - Raise application errors, if necessary.
  - Check for variables that are **NULL** or **NOT NULL**.
  - Check for queries that return no rows of data, or more rows of data than the query should return.
- Make use of the **%TYPE** and **%ROWTYPE** directives in case a column's datatype changes.
- Avoid implicit data conversions.
- Write smaller blocks of code, using procedures and functions for repetitious code.
- Storing variable-length character data can often lead to unintended results.
  - Typically, a user enters a string variable without trailing blanks.
    - When you compare the value the user supplies with the value stored in the **CHAR** datatype, you'll have to remember to use the **RPAD** function or your comparison will fail.
  - Use the **VARCHAR2** datatype instead.

 © 2011 ITCourseware, LLC

It is a good idea to use the **%TYPE** and **%ROWTYPE** directives when declaring variables that will hold data selected from a table. For instance, you might have declared a variable as follows:

```
last  VARCHAR2(15);
```

Someone could later change the **cust_last_name** column of the **customers** table to hold 30 characters of data. The following code would return an error if the **cust_last_name** column holds a value longer than 15 characters:

```
SELECT  cust_last_name
  INTO  last
  FROM  customers
 WHERE  customer_id = 101;
```

Instead, declare the variable as follows, and no errors will result:

```
last  customers.cust_last_name%TYPE;
```

LABS

❶ Modify the *multi_blocks.sql* script to contain the appropriate exception handlers in each sub-block. Use **DBMS_OUTPUT** to write an appropriate error message to the screen.
(Solution: *multi_blocks2.sql*)

❷ It has been determined that it is company policy that no raise shall exceed 10%. Modify *give_raises.sql* to raise a user-defined exception for a raise that exceeds this threshold. Run your modified script multiple times to see how the code is interrupted by your exception.
(Solution: *give_raises2.sql*)

❸ Modify *power.sql*. Add calls to **DBMS_OUTPUT** to help you follow the execution of the script. Make sure to show all variable contents in the loop.
(Solution: *power2.sql*)

 © 2011 ITCourseware, LLC

 © 2011 ITCourseware, LLC