PART I
Programming in PL/SQL

This first part of this book introduces PL/SQL, explains how to create and run PL/SQL code, and presents language fundamentals. Chapter 1 asks the fundamental questions: where did PL/SQL come from? What is it good for? What are the main features of the PL/SQL language? Chapter 2 is designed to get you and up and running PL/SQL programs as quickly as possible; it contains clear, straightforward instructions for executing PL/SQL code in SQL*Plus and a few other common environments. Chapter 3 answers basic questions about the language structure and keywords: what makes up a PL/SQL statement? What is the PL/SQL block structure all about? How do I write comments in PL/SQL?

CHAPTER 1

Introduction to PL/SQL

PL/SQL stands for "Procedural Language extensions to the Structured Query Language." SQL is the now-ubiquitous language for both querying *and* updating—never mind the name—of relational databases. Oracle Corporation introduced PL/SQL to overcome some limitations in SQL and to provide a more complete programming solution for those who sought to build mission-critical applications to run against the Oracle database. This chapter introduces PL/SQL, its origins, and its various versions. It offers a quick summary of PL/SQL in the latest Oracle release, Oracle Database 12*c*. Finally, it provides a guide to additional resources for PL/SQL developers and some words of advice.

What Is PL/SQL?

Oracle's PL/SQL language has several defining characteristics:

It is a highly structured, readable, and accessible language
: If you are new to programming, PL/SQL is a great place to start. You will find that it is an easy language to learn and is rich with keywords and structure that clearly express the intent of your code. If you are experienced in other programming languages, you will very easily adapt to the new syntax.

It is a standard and portable language for Oracle development
: If you write a PL/SQL procedure or function to execute from within the Oracle database sitting on your laptop, you can move that same procedure to a database on your corporate network and execute it there without any changes (assuming compatibility of Oracle versions, of course!). "Write once, run everywhere" was the mantra of PL/SQL long before Java appeared. For PL/SQL, though, "everywhere" means "everywhere there is an Oracle database."

It is an embedded language
: PL/SQL was not designed to be used as a standalone language, but instead to be invoked from within a host environment. So, for example, you can run PL/SQL programs from within the database (through, say, the SQL*Plus interface). Alternatively, you can define and execute PL/SQL programs from within an Oracle Developer form or report (this approach is called *client-side PL/SQL*). You cannot, however, create a PL/SQL executable that runs all by itself.

It is a high-performance, highly integrated database language
: These days, you have a number of choices when it comes to writing software to run against the Oracle database. You can use Java and JDBC; you can use Visual Basic and ODBC; you can go with Delphi, C++, and so on. You will find, however, that it is easier to write highly efficient code to access the Oracle database in PL/SQL than it is in any other language. In particular, Oracle offers certain PL/SQL-specific enhancements, such as the FORALL statement, that can improve database performance by an order of magnitude or more.

The Origins of PL/SQL

Oracle Corporation has a history of leading the software industry in providing declarative, nonprocedural approaches to designing both databases and applications. The Oracle Server technology is among the most advanced, powerful, and stable relational databases in the world. Its application development tools, such as Oracle Forms, offer high levels of productivity by relying heavily on a "paint your screen" approach in which extensive default capabilities allow developers to avoid heavy customized programming efforts.

The Early Years of PL/SQL

In Oracle's early years, the declarative approach of SQL, combined with its groundbreaking relational technology, was enough to satisfy developers. But as the industry matured, expectations rose, and requirements became more stringent. Developers needed to get "under the skin" of the products. They needed to build complicated formulas, exceptions, and rules into their forms and database scripts.

In 1988, Oracle Corporation released Oracle version 6, a major advance in its relational database technology. A key component of that version was the so-called procedural option, or PL/SQL. At roughly the same time, Oracle released its long-awaited upgrade to SQL*Forms version 2.3 (the original name for the product now known as Oracle Forms or Forms Developer). SQL*Forms v3 incorporated the PL/SQL engine for the first time on the tools side, allowing developers to code their procedural logic in a natural, straightforward manner.

This first release of PL/SQL was very limited in its capabilities. On the server side, you could use PL/SQL only to build "batch processing" scripts of procedural and SQL statements. You could not construct a modular application or store business rules in the server. On the client side, SQL*Forms v3.0 did allow you to create procedures and functions, although support for functions was not documented and therefore they were not used by many developers for years. In addition, this release of PL/SQL did not implement array support and could not interact with the operating system (for input or output). It was a far cry from a full-fledged programming language.

But for all its limitations, PL/SQL was warmly, even enthusiastically, received in the developer community. The hunger for the ability to code a simple IF statement inside SQL*Forms was strong. The need to perform multi-SQL statement batch processing was overwhelming.

What few developers realized at the time was that the original motivation and driving vision behind PL/SQL extended beyond the desire for programmatic control within products like SQL*Forms. Very early in the life cycle of Oracle's database and tools, Oracle Corporation had recognized two key weaknesses in their architecture: lack of portability and problems with execution authority.

Improved Application Portability

The concern about portability might seem odd to those of us familiar with Oracle Corporation's marketing and technical strategies. One of the hallmarks of the Oracle solution from the early 1980s was its portability. At the time that PL/SQL came along, the C-based database ran on many different operating systems and hardware platforms. SQL*Plus and SQL*Forms adapted easily to a variety of terminal configurations. Yet for all that coverage, there were still many applications that needed the more sophisticated and granular control offered by such host languages as COBOL, C, and FORTRAN. As soon as a developer stepped outside the port-neutral Oracle tools, the resulting application would no longer be portable.

The PL/SQL language was (and is) intended to widen the range of application requirements that can be handled entirely in operating system–independent programming tools. Today, Java and other programming languages offer similar portability. Yet PL/SQL stands out as an early pioneer in this field and, of course, it continues to allow developers to write highly portable application code.

Improved Execution Authority and Transaction Integrity

An even more fundamental issue than portability was execution authority. The database and the SQL language let you tightly control access to, and changes in, any particular database table. For example, with the GRANT command, you can make sure that only certain roles and users can perform an UPDATE on a given table. This GRANT command, on the other hand, cannot ensure that a user will make the correct sequence of

changes to one or more tables that are commonly needed with most business transactions.

The PL/SQL language provides tight control and management over logical transactions. One way PL/SQL does this is with the implementation of execution authority. Instead of granting to a role or user the authority to update a table, you grant privileges only to execute a procedure, which controls and provides access to the underlying data structures. The procedure is owned by a different Oracle database schema (the "definer" of the program), which, in turn, is granted the actual update privileges on those tables needed to perform the transaction. The procedure therefore becomes the "gatekeeper" for the transaction. The only way that a program (whether it's an Oracle Forms application or a Pro*C executable) can execute the transfer is through the procedure. In this way, the overall application transaction integrity is guaranteed.

Starting with Oracle8*i* Database, Oracle added considerable flexibility to the execution authority model of PL/SQL by offering the AUTHID clause. With AUTHID, you can continue to run your programs under the definer rights model described earlier, or you can choose AUTHID CURRENT_USER (invoker rights), in which case the programs run under the authority of the invoking (current) schema. Invoker rights is just one example of how PL/SQL has matured and become more flexible over the years.

Humble Beginnings, Steady Improvement

As powerful as SQL is, it simply does not offer the flexibility and power developers need to create full-blown applications. Oracle's PL/SQL language ensures that we can stay entirely within the operating system–independent Oracle environment and still write highly efficient applications that meet our users' requirements.

PL/SQL has come a long way from its humble beginnings. With PL/SQL 1.0, it was not uncommon for developers to have to tell their managers, "You can't do that with PL/SQL." Today, that statement has moved from fact to excuse. If you are ever confronted with a requirement and find yourself saying, "There's no way to do that," please don't repeat it to your manager. Instead, dig deeper into the language, or explore the range of PL/SQL packages offered by Oracle. It is extremely likely that PL/SQL today will, in fact, allow you to do pretty much whatever you need to do.

Over the years, Oracle Corporation has demonstrated its commitment to PL/SQL, its flagship proprietary programming language. With every new release of the database, Oracle has also made steady, fundamental improvements to the PL/SQL language itself. It has added a great variety of supplied (or *built-in*) packages that extend the PL/SQL language in numerous ways and directions. It has introduced object-oriented capabilities, implemented a variety of array-like data structures, enhanced the compiler to both optimize our code and provide warnings about possible quality and performance issues, and in general improved the breadth and depth of the language.

The next section presents some examples of PL/SQL programs that will familiarize you with the basics of PL/SQL programming.

So This Is PL/SQL

If you are completely new to programming or to working with PL/SQL (or even SQL, for that matter), learning PL/SQL may seem an intimidating prospect. If this is the case, don't fret! I am confident that you will find it easier than you think. There are two reasons for my optimism:

- Computer languages in general are not that hard to learn, at least compared to a second or third human language. The reason? It's simply that computers are not particularly smart (they "think"—perform operations—rapidly, but not at all creatively). We must rely on a very rigid syntax in order to tell a computer what we want it to do. So the resulting language is also rigid (no exceptions!) and therefore easier for us to pick up.
- PL/SQL truly is an easy language, compared to other programming languages. It relies on a highly structured "block" design with different sections, all identified with explicit, self-documenting keywords.

Let's look at a few examples that demonstrate some key elements of both PL/SQL structure and functionality.

Integration with SQL

One of the most important aspects of PL/SQL is its tight integration with SQL. You don't need to rely on any intermediate software "glue" such as ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) to run SQL statements in your PL/SQL programs. Instead, you just insert the UPDATE or SELECT into your code, as shown here:

```
 1  DECLARE
 2     l_book_count INTEGER;
 3
 4  BEGIN
 5     SELECT COUNT(*)
 6       INTO l_book_count
 7       FROM books
 8      WHERE author LIKE '%FEUERSTEIN, STEVEN%';
 9
10     DBMS_OUTPUT.PUT_LINE (
11        'Steven has written (or co-written) ' ||
12         l_book_count ||
13         ' books.');
14
15     -- Oh, and I changed my name, so...
```

```
16     UPDATE books
17        SET author = REPLACE (author, 'STEVEN', 'STEPHEN')
18      WHERE author LIKE '%FEUERSTEIN, STEVEN%';
19  END;
```

Let's take a more detailed look at this code in the following table.

Line(s)	Description
1–3	This is the declaration section of this so-called "anonymous" PL/SQL block, in which I declare an integer variable to hold the number of books that I have authored or coauthored. (I'll say much more about the PL/SQL block structure in Chapter 3.)
4	The BEGIN keyword indicates the beginning of my execution section—the code that will be run when I pass this block to SQL*Plus.
5–8	I run a query to determine the total number of books I have authored or coauthored. Line 6 is of special interest: the INTO clause shown here is actually not part of the SQL statement but instead serves as the bridge from the database to local PL/SQL variables.
10–13	I use the DBMS_OUTPUT.PUT_LINE built-in procedure (i.e., a procedure in the DBMS_OUTPUT package supplied by Oracle) to display the number of books.
15	This single-line comment explains the purpose of the UPDATE.
16–18	I have decided to change the spelling of my first name to "Stephen", so I issue an update against the books table. I take advantage of the built-in REPLACE function to locate all instances of "STEVEN" and replace them with "STEPHEN".

Control and Conditional Logic

PL/SQL offers a full range of statements that allow us to very tightly control which lines of our programs execute. These statements include:

IF and CASE statements
: These implement conditional logic; for example, "If the page count of a book is greater than 1,000, then..."

A full complement of looping or iterative controls
: These include the FOR loop, the WHILE loop, and the simple loop.

The GOTO statement
: Yes, PL/SQL even offers a GOTO that allows you to branch unconditionally from one part of your program to another. That doesn't mean, however, that you should actually *use* it.

Here is a procedure (a reusable block of code that can be called by name) that demonstrates some of these features:

```
1  PROCEDURE pay_out_balance (
2     account_id_in IN accounts.id%TYPE)
3  IS
4     l_balance_remaining NUMBER;
5  BEGIN
6     LOOP
```

```
 7        l_balance_remaining := account_balance (account_id_in);
 8
 9        IF l_balance_remaining < 1000
10        THEN
11           EXIT;
12        ELSE
13           apply_balance (account_id_in, l_balance_remaining);
14        END IF;
15     END LOOP;
16  END pay_out_balance;
```

Let's take a more detailed look at this code in the following table.

Line(s)	Description
1–2	This is the header of a procedure that pays out the balance of an account to cover outstanding bills. Line 2 is the parameter list of the procedure, in this case consisting of a single incoming value (the identification number of the account).
3–4	This is the declaration section of the procedure. Notice that instead of using a DECLARE keyword, as in the previous example, I use the keyword IS (or AS) to separate the header from the declarations.
6–15	Here is an example of a simple loop. This loop relies on an EXIT statement (see line 11) to terminate the loop; FOR and WHILE loops specify the termination condition differently.
7	Here, I call to the account_balance function to retrieve the balance for this account. This is an example of a call to a reusable program within another reusable program. Line 13 demonstrates the calling of another procedure within this procedure.
9–14	Here is an IF statement that can be interpreted as follows: if the account balance has fallen below $1,000, stop allocating funds to cover bills. Otherwise, apply the balance to the next charge.

When Things Go Wrong

The PL/SQL language offers a powerful mechanism for both raising and handling errors. In the following procedure, I obtain the name and balance of an account from its ID. I then check to see if the balance is too low. If it is, I explicitly raise an exception, which stops my program from continuing:

```
 1  PROCEDURE check_account (
 2     account_id_in IN accounts.id%TYPE)
 3  IS
 4     l_balance_remaining       NUMBER;
 5     l_balance_below_minimum   EXCEPTION;
 6     l_account_name            accounts.name%TYPE;
 7  BEGIN
 8     SELECT name
 9       INTO l_account_name
10       FROM accounts
11      WHERE id = account_id_in;
12
13     l_balance_remaining := account_balance (account_id_in);
14
15     DBMS_OUTPUT.PUT_LINE (
```

```
16         'Balance for ' || l_account_name ||
17          ' = ' || l_balance_remaining);
18
19      IF l_balance_remaining < 1000
20      THEN
21         RAISE l_balance_below_minimum;
22      END IF;
23
24   EXCEPTION
25      WHEN NO_DATA_FOUND
26      THEN
27         -- No account found for this ID
28         log_error (...);
29         RAISE;
30      WHEN l_balance_below_minimum
31      THEN
32         log_error (...);
33         RAISE VALUE_ERROR;
34   END;
```

Let's take a more detailed look at the error-handling aspects of this code in the following table.

Line(s)	Description
5	I declare my own exception, called l_balance_below_minimum. Oracle provides a set of predefined exceptions, such as DUP_VAL_ON_INDEX, but I need something specific to my application, so I must define it myself in this case.
8–11	This query retrieves the name for the account. If there is no account for this ID, the database raises the predefined NO_DATA_FOUND exception, causing the program to stop.
19–22	If the balance is too low, I explicitly raise my own exception because I have encountered a serious problem with this account.
24	The EXCEPTION keyword denotes the end of the executable section and the beginning of the exception section in which errors are handled.
25–28	This is the error-handling section for the situation in which the account is not found. If NO_DATA_FOUND was the exception raised, it is trapped here, and the error is logged with the log_error procedure. I then re-raise the *same exception*, so the outer block will be aware that there was no match for that account ID.
30–33	This is the error-handling section for the situation in which the account balance has gotten too low (my application-specific exception). If l_balance_below_minimum is raised, it's trapped here, and the error is logged. I then raise the system-defined VALUE_ERROR exception to notify the outer block of the problem.

Chapter 6 takes you on an extensive tour of PL/SQL's error-handling mechanisms.

There is, of course, much more that can be said about PL/SQL—which is why you have hundreds more pages of material to study in this book! These initial examples should, however, give you a feel for the kind of code you will write with PL/SQL, some of its most important syntactical elements, and the ease with which one can write—and read—PL/SQL code.

About PL/SQL Versions

Each version of the Oracle database comes with its own corresponding version of PL/SQL. As you use more up-to-date versions of PL/SQL, an increasing array of functionality will be available to you. One of our biggest challenges as PL/SQL programmers is simply keeping up. We need to constantly educate ourselves about the new features in each version—figuring out how to use them and how to apply them to our applications, and determining which new techniques are so useful that we should modify existing applications to take advantage of them.

Table 1-1 summarizes the major elements in each of the versions (past and present) of PL/SQL in the database. (Note that in early versions of the database, PL/SQL version numbers differed from database release numbers, but since Oracle8 Database, they have been identical.) The table offers a very high-level glimpse of the new features available in each version. Following the table, you will find more detailed descriptions of "what's new" in PL/SQL in the latest Oracle version, Oracle Database 12*c*.

The Oracle Developer product suite also comes with its own version of PL/SQL, and it generally lags behind the version available in the Oracle database itself. This chapter (and the book as a whole) concentrates on server-side PL/SQL programming.

Table 1-1. Oracle Database and corresponding PL/SQL versions

Oracle Database release	PL/SQL version highlights
6.0	The initial version of PL/SQL (1.0) was used primarily as a scripting language in SQL*Plus (it was not yet possible to create named, reusable, and callable programs) and also as a programming language in SQL*Forms 3.
7.0	This major upgrade (2.0) to PL/SQL 1.0 added support for stored procedures, functions, packages, programmer-defined records, PL/SQL tables (now known as collections), and many package extensions.
7.1	This PL/SQL version (2.1) supported programmer-defined subtypes, enabled the use of stored functions inside SQL statements, and offered dynamic SQL with the DBMS_SQL package. With PL/SQL 2.1, you could execute SQL DDL statements from within PL/SQL programs.
7.3	This PL/SQL version (2.3) provided enhanced functionality of collections, offered improved remote dependency management, added file I/O capabilities to PL/SQL with the UTL_FILE package, and completed the implementation of cursor variables.
8.0	The new version number (8.0) for PL/SQL reflected Oracle's effort to synchronize version numbers across related products. PL/SQL 8.0 is the version of PL/SQL that supported enhancements of Oracle8 Database, including large objects (LOBs), object-oriented design and development, collections (VARRAYs and nested tables), and the Oracle/Advanced Queuing facility (Oracle/AQ).
8.1	The first of Oracle's *i* series; the corresponding release of PL/SQL offered a truly impressive set of added functionality, including a new version of dynamic SQL, support for Java in the database, the invoker rights model, the execution authority option, autonomous transactions, and high-performance "bulk" DML and queries.

Oracle Database release	PL/SQL version highlights
9.1	Oracle9*i* Database Release 1 came fairly quickly on the heels of its predecessor. The first release of this version included support for inheritance in object types, table functions and cursor expressions (allowing for parallelization of PL/SQL function execution), multilevel collections, and the CASE statement and CASE expression.
9.2	Oracle9*i* Database Release 2 put a major emphasis on XML (Extensible Markup Language) but also had some treats for PL/SQL developers, including associative arrays that can be indexed by VARCHAR2 strings in addition to integers, record-based DML (allowing you to perform an insert using a record, for example), and many improvements to UTL_FILE (which allows you to read/write files from within a PL/SQL program).
10.1	Oracle Database 10*g* Release 1 was unveiled in 2004 and focused on support for grid computing, with an emphasis on improved/automated database management. From the standpoint of PL/SQL, the most important new features, an optimized compiler and compile-time warnings, were transparently available to developers.
10.2	Oracle Database 10*g* Release 2, released in 2005, offered a small number of new features for PL/SQL developers, most notably support for preprocessor syntax that allows you to conditionally compile portions of your program, depending on Boolean expressions you define.
11.1	Oracle Database 11*g* Release 1 arrived in 2007. The most important feature for PL/SQL developers was the function result cache, but there are also some other goodies like compound triggers, the CONTINUE statement, and native compilation that produces machine code.
11.2	Oracle Database 11*g* Release 2 became available in the fall of 2009; the most important new feature overall is the edition-based redefinition capability, which allows administrators to "hot patch" applications while they are being executed by users.
12.1	Oracle Database 12*c* Release 1 became available in June 2013; it offers several enhancements for managing access to and privileges on program units and views; brings the SQL and PL/SQL languages more into sync, particularly regarding maximum VARCHAR2 lengths and dynamic SQL binding; supports the definition of simple functions within SQL statements; and adds the UTL_CALL_STACK package, for fine-grained access to the execution call stack, error stack, and error backtrace.

Oracle Database 12c New PL/SQL Features

Oracle Database 12*c* offers a number of new features that improve the performance and usability of PL/SQL. It also rounds out some rough edges of the language. Here is a summary of the most important changes for PL/SQL developers.

More PL/SQL-only datatypes cross PL/SQL-to-SQL interface

Prior to 12.1, you could not bind PL/SQL-specific datatypes (for example, an associative array) in a dynamic SQL statement. Now it is possible to bind values with PL/SQL-only datatypes in anonymous blocks, PL/SQL function calls in SQL queries, CALL statements, and the TABLE operator in SQL queries.

ACCESSIBLE_BY clause

You can now include an ACCESSIBLE_BY clause in your package specification, specifying which program units may invoke subprograms in the package. This feature allows

you to "expose" subprograms in helper packages that are intended to be consumed only by specific program units. This feature offers a kind of "whitelisting" for packages.

Implicit statement results

Before Oracle Database 12*c*, a PL/SQL stored subprogram returned result sets from SQL queries explicitly, through an OUT REF CURSOR parameter or RETURN clause. The client program would then have to bind to those parameters explicitly to receive the result sets. Now, a PL/SQL stored subprogram can return query results to its client implicitly, using the PL/SQL package DBMS_SQL instead of OUT REF CURSOR parameters. This functionality will make it easy to migrate applications that rely on the implicit return of query results from stored subprograms (supported by languages like Transact SQL) from third-party databases to Oracle Database.

BEQUEATH CURRENT_USER views

Before Oracle Database 12*c*, a view always behaved like a definer rights unit (AUTHID DEFINER), even if it was referenced inside an invoker rights unit (AUTHID CURRENT_USER). Now, a view can be either BEQUEATH DEFINER (the default), which behaves like a definer rights unit, or BEQUEATH CURRENT_USER, which behaves *somewhat like* an invoker rights unit.

Grant roles to program units

Prior to Oracle Database 12*c*, an invoker rights unit always ran with the privileges of its invoker. If its invoker had *higher* privileges than its owner, then the invoker rights unit might perform operations unintended by, or forbidden to, its owner.

As of 12.1, you can grant roles to individual PL/SQL packages and standalone subprograms. Instead of a definer rights unit, you can create an invoker rights unit and then grant roles to it. The invoker rights unit then runs with the privileges of both the invoker and the roles, but without any additional privileges possessed by the definer's schema.

An invoker rights unit can now run with the privileges of its invoker only if its owner has either the INHERIT PRIVILEGES privilege on the invoker or the INHERIT ANY PRIVILEGES privilege.

The INHERIT PRIVILEGES privilege is granted to all schemas on install/upgrade.

New conditional compilation directives

In 12.1, Oracle has added two new predefined inquiry directives, $$PLSQL_UNIT_OWNER and $$PLSQL_UNIT_TYPE, which return the owner and type of the current PL/SQL program unit.

Optimizing function execution in SQL

Oracle now offers two ways of improving PL/SQL function performance in SQL statements: you can actually *define* the function itself inside the SQL statement using the WITH clause, or you can add the UDF pragma to the program unit, which tells the compiler that the function will be used primarily in SQL statements.

Using %ROWTYPE with invisible columns

Oracle Database 12*c* makes it possible to define *invisible* columns. From within PL/SQL, the %ROWTYPE attribute is aware of these types of columns and how to work with them.

FETCH FIRST clause and BULK COLLECT

In 12.1, you can use the optional FETCH FIRST clause to limit the number of rows that a query returns, significantly reducing the SQL complexity of common "Top *N*" queries. FETCH FIRST will be of most benefit in the simplification of migration from third-party databases to Oracle Database. This clause can also, however, improve the performance of some SELECT BULK COLLECT INTO statements.

The UTL_CALL_STACK package

Prior to 12.1, the DBMS_UTILITY package offered three functions (FORMAT_CALL_STACK, FORMAT_ERROR_STACK, and FORMAT_ERROR_BACKTRACE) to provide information about the execution call stack, error stack, and error backtrace, respectively. In 12.1, a single package, UTL_CALL_STACK, provides that same information, plus much more fine-grained access to the contents of these formatted strings.

Resources for PL/SQL Developers

O'Reilly published the first edition of this book back in 1995. At that time, *Oracle PL/SQL Programming* made quite a splash. It was the first independent (i.e., not emanating from Oracle) book on PL/SQL, and it fulfilled a clear and intensely felt need of developers around the world. Since that time, resources—books, development environments, utilities, and websites—for PL/SQL programmers have proliferated. (Of course, this book is still by far the most important and valuable of these resources!)

The following sections describe very briefly many of these resources. By taking full advantage of these resources, many of which are available either for free or at a relatively low cost, you will greatly improve your development experience (and resulting code).

The O'Reilly PL/SQL Series

Over the years, the Oracle PL/SQL series from O'Reilly has grown to include quite a long list of books. Here we've summarized the books currently in print. Please check out the Oracle area of the O'Reilly website for much more complete information.

Oracle PL/SQL Programming, by Steven Feuerstein with Bill Pribyl
: The 1,300-page tome you are reading now. The desk-side companion of a great many professional PL/SQL programmers, this book is designed to cover every feature in the core PL/SQL language. The current version covers through Oracle Database 11*g* Release 2.

Learning Oracle PL/SQL, by Bill Pribyl with Steven Feuerstein
: A comparatively gentle introduction to the language, ideal for new programmers and those who know a language other than PL/SQL.

Oracle PL/SQL Best Practices, by Steven Feuerstein
: A relatively short book that describes dozens of best practices that will help you produce high-quality PL/SQL code. Having this book is kind of like having a "lessons learned" document written by an in-house PL/SQL expert. The second edition features completely rewritten content that teaches best practices by following the challenges of a development team writing code for the make-believe company My Flimsy Excuse.

Oracle PL/SQL Developer's Workbook, by Steven Feuerstein with Andrew Odewahn
: Contains a series of questions and answers intended to help PL/SQL programmers develop and test their understanding of the language. This book covers PL/SQL features through Oracle8*i* Database, but of course most of those exercises apply to later versions of the database as well.

Oracle Built-in Packages, by Steven Feuerstein, Charles Dye, and John Beresniewicz
: A reference guide to the prebuilt packages that Oracle supplies with the core database server. The use of these packages can often simplify the difficult and tame the impossible. This book covers features through Oracle8 Database, but the in-depth explanations of and examples for the included packages are still very helpful in later releases.

Oracle PL/SQL for DBAs , by Arup Nanda and Steven Feuerstein
: The PL/SQL language becomes more important to Oracle DBAs with each new version of the database. There are two main reasons for this. First, large amounts of DBA functionality are made available through a PL/SQL package API. To use this functionality, you must also write and run PL/SQL programs. Second, it is

critical that DBAs have a working knowledge of PL/SQL so that they can identify problems in the code built by developers. This book offers a wealth of material that will help DBAs get up to speed quickly so they can fully leverage PL/SQL to get their jobs done.

Oracle PL/SQL Language Pocket Reference, by Steven Feuerstein, Bill Pribyl, and Chip Dawes

A small but very useful quick-reference book that might actually fit in your coat pocket. It summarizes the syntax of the core PL/SQL language through Oracle Database 11*g*.

Oracle PL/SQL Built-ins Pocket Reference, by Steven Feuerstein, John Beresniewicz, and Chip Dawes

Another helpful and concise guide summarizing built-in functions and packages through Oracle8 Database.

PL/SQL on the Internet

There are also many online resources for PL/SQL programmers. This list focuses primarily on those resources for which the coauthors provide or manage content.

Steven Feuerstein's PL/SQL Obsession

PL/SQL Obsession is Steven's online portal for PL/SQL resources, including all of his training presentations and supporting code, freeware utilities (some listed here), video recordings, and more.

PL/SQL Challenge

The PL/SQL Challenge is a website that promotes "active learning" — rather than passively reading a book or web page, you take quizzes on PL/SQL, SQL, logic, Database Design, and Oracle Application Express, thereby testing your knowledge.

PL/SQL Channel

The PL/SQL Channel offers a library of over 27 hours of video training on the Oracle PL/SQL language, all recorded by Steven Feuerstein.

Oracle Technology Network

The Oracle Technology Network (OTN) "provides services and resources that developers need to build, test, and deploy applications" based on Oracle technology. Boasting membership in the millions, OTN is a great place to download Oracle software, documentation, and lots of sample code. PL/SQL also has its own page on the OTN website.

Quest Error Manager

The Quest Error Manager (QEM) is a framework that will help you standardize the management of errors in a PL/SQL-based application. With QEM, you can register, raise, and report on errors through an API that makes it easy for all developers to

perform error management in the same way, with a minimum amount of effort. Error information is logged into the instance (general information about the error) and context (application-specific name/value pairs) tables.

oracle-developer.net
: Maintained by Adrian Billington (Who wrote the section in Chapter 21 on pipelined table functions), this site is a resource for Oracle database developers which contains an outstanding collection of articles, tutorials, and utilities. Adrian offers in-depth treatments of new features in each release of Oracle Database, full of examples, performance analysis scripts, and more.

ORACLE-BASE
: ORACLE-BASE is another fantastic resource for Oracle technologists built and maintained by a single Oracle expert: Tim Hall. Tim is an Oracle ACE Director, OakTable Network member, and was chosen as Oracle ACE of the Year 2006 by Oracle Magazine Editor's Choice Awards. He has been invovled in DBA, design, and development work with Oracle databases since 1994. See http://oracle-base.com.

Some Words of Advice

Since 1995, when the first edition of this book was published, I have had the opportunity to train, assist, and work with tens of thousands of PL/SQL developers. In the process, I have learned an awful lot and have also gained some insights into the way we all do our work in the world of PL/SQL. I hope that you will not find it too tiresome if I share some advice with you on how you can work more effectively with this powerful programming language.

Don't Be in Such a Hurry!

We are almost always working under tight deadlines, or playing catch-up from one setback or another. We have no time to waste, and lots of code to write. So let's get right to it—right?

Wrong. If we dive too quickly into the depths of code construction, slavishly converting requirements to hundreds, thousands, or even tens of thousands of lines of code, we will end up with a total mess that is almost impossible to debug and maintain. Don't respond to looming deadlines with panic; you are more likely to meet those deadlines if you do some careful planning.

I strongly encourage you to resist these time pressures and make sure to do the following before you start a new application, or even a specific program in an application:

Construct test cases and test scripts before you write your code
: You should determine how you want to verify a successful implementation before you write a single line of a program. If you do this, you are more likely to get the interface of your programs correct, as it will help you thoroughly identify what it is your program needs to do.

Establish clear rules for how developers will write the SQL statements in the application
: In general, I recommend that individual developers not write a whole lot of SQL. Instead, those single-row queries and inserts and updates should be "hidden" behind prebuilt and thoroughly tested procedures and functions (this is called *data encapsulation*). These programs can be optimized, tested, and maintained much more effectively than SQL statements (many of them redundant) scattered throughout your code.

Establish clear rules for how developers will handle exceptions in the application
: All developers on a team should raise, handle, and log errors in the same way. The best way to do this is to create a single error-handling package that hides all the details of how an error log is kept, determines how exceptions are raised and propagated up through nested blocks, and avoids hardcoding of application-specific exceptions. Make sure that all developers use this package and that they do *not* write their own complicated, time-consuming, and error-prone error-handling code.

Use top-down design (a.k.a. stepwise refinement) to limit the complexity of the requirements you must deal with at any given time
: If you use this approach, you will find that the executable sections of your modules are shorter and easier to understand, which makes your code easier to maintain and enhance over time. Using local or nested modules plays a key role in following this design principle.

These are just a few of the important things to keep in mind *before* you start writing all that code. Just remember: in the world of software development, haste not only makes waste, but virtually guarantees a generous offering of bugs and lost weekends.

Don't Be Afraid to Ask for Help

Chances are, if you are a software professional, you are a fairly smart individual. You studied hard, you honed your skills, and now you make a darn good living writing code. You can solve almost any problem you are handed, and that makes you proud. Unfortunately, your success can also make you egotistical, arrogant, and reluctant to seek out help when you are stumped. This dynamic is one of the most dangerous and destructive aspects of software development.

Software is written by human beings; it is important, therefore, to recognize that human psychology plays a key role in software development. The following is an example.

Joe, the senior developer in a team of six, has a problem with his program. He studies it for hours, with increasing frustration, but he cannot figure out the source of the bug. He wouldn't think of asking any of his peers to help because they all have less experience than he does. Finally, though, he is at his wits' end and "gives up." Sighing, he picks up his phone and touches an extension: "Sandra, could you come over here and take a look at my program? I've got a problem I simply cannot figure out." Sandra stops by and, with the quickest glance at Joe's program, points out what should have been obvious to him long ago. Hurray! The program is fixed, and Joe expresses gratitude, but in fact he is secretly embarrassed.

Thoughts like "Why didn't I see that?" and "If I'd only spent another five minutes doing my own debugging I would have found it" run though Joe's mind. This is understandable, but also very thick-headed. The bottom line is that we are often unable to identify our own problems because we are too close to our own code. Sometimes, all we need is a fresh perspective, the relatively objective view of someone with nothing at stake. It has nothing to do with seniority, expertise, or competence.

We strongly suggest that you establish the following guidelines in your organization:

Reward admissions of ignorance
: Hiding what you don't know about an application or its code is very dangerous. Develop a culture that welcomes questions and requests for help.

Ask for help
: If you cannot figure out the source of a bug in 30 minutes, immediately ask for help. You might even set up a "buddy system," so that everyone is assigned a person who is *expected* to be asked for assistance. Don't let yourself (or others in your group) spend hours banging your head against the wall in a fruitless search for answers.

Set up a formal peer code review process
: Don't let any code go to QA or production without being read and critiqued (in a positive, constructive manner) by one or more other developers in your group.

Take a Creative, Even Radical Approach

We all tend to fall into ruts, in almost every aspect of our lives. People are creatures of habit: you learn to write code in one way; you assume certain limitations about a product; you turn aside possible solutions without serious examination because you just *know* it cannot be done. Developers become downright prejudiced about their own programs, and often not in positive ways. They are often overheard saying things like:

- "It can't run any faster than that; it's a pig."
- "I can't make it work the way the user wants; that'll have to wait for the next version."

- "If I were using X or Y or Z product, it would be a breeze. But with this stuff, everything is a struggle."

But the reality is that your program can almost always run a little faster. And the screen can, in fact, function *just* the way the user wants it to. And although each product has its limitations, strengths, and weaknesses, you should never have to wait for the next version. Isn't it so much more satisfying to be able to tell your therapist that you tackled the problem head-on, accepted no excuses, and crafted a solution?

How do you do this? Break out of the confines of your hardened views and take a fresh look at the world (or maybe just your cubicle). Reassess the programming habits you've developed. Be creative—step away from the traditional methods, from the often limited and mechanical approaches constantly reinforced in our places of business.

Try something new: experiment with what may seem to be a radical departure from the norm. You will be surprised at how much you will learn and grow as a programmer and problem solver. Over the years, I have surprised myself over and over with what is really achievable when I stop saying, "You can't do that!" and instead simply nod quietly and murmur, "Now, if I do it this way..."