CHAPTER 25

Globalization and Localization in PL/SQL

Businesses do not typically begin their operations on a global scale. They usually start as local or regional businesses with plans to expand. As they grow into new regions, or locales, critical applications need to adjust to new language and formatting requirements. If the applications are not designed to support multiple locales, this transition becomes very time-consuming and costly.

In a perfect development world, globalization would be an integral part of application design, and all design considerations would be accompanied by the question, "Where in the world would this design fail?" In the real world, however, many companies do not include a globalization strategy in the initial design. Cost concerns, lack of globalization experience, or simply an inability to anticipate the global reach of the business are all common reasons for failing to analyze localization risks.

So what's the big deal? It is just a matter of translating data, right? Not exactly. There are, in fact, many common PL/SQL programming tasks with localization implications that can disrupt your application's ability to function globally:

Variable precision
: CHAR(1) handles *F* very nicely, but will it do the same for 民?

String sort order of result set
: ORDER BY is determined easily for English characters. Will Korean, Chinese, or Japanese be as straightforward? How are combined characters, or characters with accents, ordered?

Information retrieval (IR)
: PL/SQL is often the language of choice for information retrieval applications. How can you store data in multiple languages and accurately retrieve information across languages using the same query?

Date/time formatting
: Different countries and regions use different calendars and date formats. How vulnerable is your code to these variations?

Currency formatting
: Currency considerations extend beyond basic currency conversion. A misused comma or period resulting from locale-specific formatting requirements can unintentionally change the cost of an item.

This chapter explores the ramifications of these kinds of issues and talks about how you can write PL/SQL code to anticipate and handle them. I begin with a discussion of Unicode and Oracle's Globalization Support architecture. Next, I demonstrate some problems using multibyte characters and describe how to handle them in your PL/SQL programs using character semantics. I then discuss some of the complexities associated with sorting character strings using various character sets, and demonstrate efficient multilingual information retrieval. Finally, I show how you can make your applications work using different date/time and currency formats.

Globalization Strategy

As you develop with globalization in mind, you will find yourself anticipating localization problems much earlier in the development life cycle. There is no better time to think about globalization than during the design phase of your application. There is obviously far more to your overall globalization strategy than can be covered in this chapter, but your PL/SQL programs should be in good shape if you take into account:

- Character set
- NLS parameters
- Unicode functions
- Character versus byte semantics
- String sort order
- Multilingual information retrieval
- Date/time
- Currency

For additional information on globalization and localization within Oracle, see the Globalization Support section of the Oracle website. Here, you will find discussion forums, links to papers, and additional documentation on database and application server globalization. There's also a Character Set Scanner download to assist with conversions.

Throughout this chapter, we will be working with a publication schema called g11n. If you want to install this catalog in your own environment, download the *G11N.ZIP* file from the book's website and unzip it to a local directory. Modify the header of *g11n.sql* with the correct environment details for your system, but *make sure when saving it again that your encoding is Unicode*. If the files are saved as ASCII or some other Western character set, the multibyte characters in the file will not save correctly and you will get errors when running the scripts. The *g11n.sql* script creates a user named g11n, grants the necessary permissions to work through the samples, creates the sample objects, and adds seed data. Refer to the header of *g11n.sql* for additional instructions.

Overview and Terminology

Before we proceed, let's get some terminology straight. Globalization, internationalization, and localization are often used interchangeably, yet they actually have very different meanings. Table 25-1 defines each and explains how they are related.

Table 25-1. Fundamental terms and abbreviation

Term	Abbreviation	Definition
Globalization	g11n	Application development strategy focused on making applications multilingual and locale-independent. Globalization is accomplished through internationalization and localization.
Internationalization	i18n	The design or modification of an application to work with multiple locales.
Localization	l10n	The process of actually making an application work in each specific locale. l10n includes text translation. It is made easier with a proper i18n implementation.

If you haven't seen the abbreviations mentioned in Table 25-1 before, you may be confused about the numbers sandwiched between the two letters. These abbreviations include the first letter and last letter of each term, with the number of characters between them in the middle. Globalization, for example, has 11 letters between the *g* and the *n*, making the abbreviation g11n.

It is true that Oracle supports localization to every region of the world. I have heard it suggested, though, that Oracle's localization support means that you can load English data and search it in Japanese. Not true! Oracle does not have a built-in linguistic translation engine that performs translations on the fly for you. If you have ever witnessed the results of a machine translation, you know that you would not want this kind of so-called functionality as a built-in "feature" anyway. Oracle *supports* localization, but it does not implement localization for you. That is still your job.

Additional terms used in this chapter are defined in Table 25-2; I'll expand on these in the following sections.

Table 25-2. Detailed globalization, localization, and internationalization terms

Term	Definition
Character encoding	Each character is a representation of a code point. Character encoding is the mapping between character and code point. The type of character encoding chosen for the database determines the ability to store and retrieve these code points.
Character set	Characters are grouped by language or region. Each regionalized set of characters is referred to as a character set.
Code point	Each character in every character set is given a unique identifier called a code point. This identifier is determined by the Unicode Consortium. Code points can represent a character in its entirety or can be combined with other code points to form complex characters. An example of a code point is \0053.
Glyph	A glyph is the graphical display of a character that is mapped to one or more code points. The code point definition in this table used the \0053 code point. The glyph this code point is mapped to is the capital letter *S*.
Multibyte characters	Most Western European characters require only a single byte to store them. Multibyte characters, such as Japanese or Korean, require between 2 and 4 bytes to store a single character in the database.
NLS	National Language Support is the old name for Oracle's globalization architecture. Starting with Oracle9*i* Database it is officially referred to as Globalization Support, but you will see documentation and parameters refer to NLS for some time to come.
Unicode	Unicode is a standard for character encoding.

Unicode Primer

Before the Unicode standard was developed, there were multiple character encoding schemes that were inadequate and that, at times, conflicted with each other. It was nearly impossible to develop global applications that were consistent because no single character encoding scheme could support all characters.

Unicode is a standard for character encoding that resolves these problems. It was developed and is maintained by the Unicode Consortium. The Unicode Standard and Unicode Character Database, or UCD, define what is included in each version.

Oracle's Unicode character sets allow you to store and retrieve more than 200 different individual character sets. Using a Unicode character set provides support for all character sets without making any engineering changes to an application.

Oracle Database 11*g* supports Unicode version 5.0. First published in 2006, Unicode 5.0 includes the capacity to encode more than 1 million characters. This is enough to support all modern characters, as well as many ancient or minor scripts. Oracle Database 12*c* includes support for Unicode 6.1 (published in January 2012) and introduces new linguistic collations complying with the Unicode Collation Algorithm (UCA).

Unicode character sets in Oracle Database 11*g* and 12*c* include UTF-8 and UTF-16 encodings. UTF-8 stores characters in 1, 2, or 3 bytes, depending on the character.

UTF-16 stores all characters in 2 bytes. Supplementary characters are supported with both encoding schemes, and these require 4 bytes per character regardless of the Unicode character set chosen.

Each Oracle database has two character sets. You can define one primary database character set that will be used for most application functions, and a separate NLS character set for NLS-specific datatypes and functions. Use the following query to determine the character sets you are using:

```
SELECT parameter, VALUE
  FROM nls_database_parameters
 WHERE parameter IN ('NLS_CHARACTERSET', 'NLS_NCHAR_CHARACTERSET')
```

This query returns the following results in my environment:

```
PARAMETER                 VALUE
------------------------- ----------
NLS_CHARACTERSET          AL32UTF8
NLS_NCHAR_CHARACTERSET    AL16UTF16
```

My NLS_CHARACTERSET, or the primary character set for my database, has a value of AL32UTF8. This 32-bit UTF-8 Unicode character set is meant to encompass most common characters in the world. My NLS_NCHAR_CHARACTERSET, used primarily for NCHAR and NVARCHAR2 columns, is a 16-bit UTF-16 character set.

Choosing a Character Set

Oracle now recommends that all new installations of the Oracle Database use a Unicode character set for the NLS_CHARACTERSET. Having performed a number of character set migrations, I agree that this recommendation is definitely a good one to follow. Your application may need to support only ASCII characters right now, but what about in two or three years? The performance implications of using Unicode are negligible, and space implications are minor since the encoding uses variable byte sizes based on the characters themselves.

Another consideration, even if you have no plans to work with multilingual data, is that you may still get multibyte characters in your database. Browser-based applications often support the copying and pasting of large amounts of text from word-processing applications. In doing so, they can take in more than simple ASCII characters. Bullets, for example, are multibyte characters. Unless you analyze everything that is posted to your data fields, it will be difficult to know whether your non-Unicode character set will support the data going in. Unicode ensures that your database will handle whatever characters are required today—and tomorrow.

The names Oracle gives to its character sets are structured to provide useful information about each character set. US7ASCII supports U.S. English characters, for example. For

AL32UTF8, the character set is intended to support *all* languages. The second part of the string indicates the number of bits per character. US7ASCII uses 7 bits per character, while AL32UTF8 uses up to 32 bits per character. The remainder of the string is the official character set name. Figure 25-1 illustrates this convention.

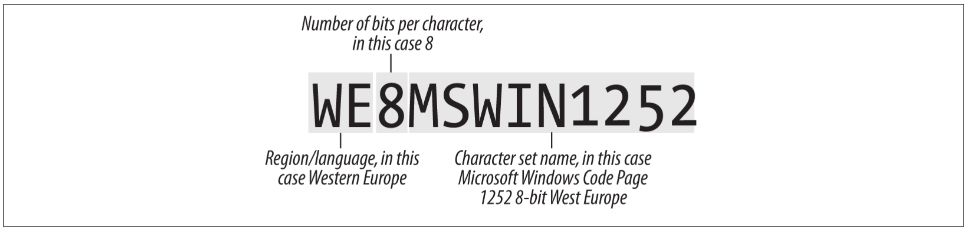
Number of bits per character, in this case 8
WE8MSWIN1252
Region/language, in this case Western Europe
Character set name, in this case Microsoft Windows Code Page 1252 8-bit West Europe

Figure 25-1. Oracle's character set naming convention

For more information on Unicode, refer to the Unicode Standard website.

National Character Set Datatypes

The Globalization Support (national character set) datatypes of NCLOB, NCHAR, and NVARCHAR2 use the character set defined for NLS_NCHAR_CHARACTERSET rather than the default character set specified for the database using NLS_CHARACTERSET. These datatypes support only the use of a multibyte Unicode character set, so even when you're working with a database whose default is non-Unicode, they will store the characters using the national character set instead. Because the national character set supports only UTF-8 and UTF-16 encodings, NCLOB, NCHAR, and NVARCHAR2 are guaranteed to store the data as multibyte Unicode.

This used to cause a problem with comparing NCLOB/NCHAR/NVARCHAR2 columns to CLOB/CHAR/VARCHAR2 columns. For all currently supported releases, however, Oracle performs an implicit conversion, allowing the comparison to take place.

Character Encoding

Your choice of a character set at database creation time determines the type of encoding your characters will have. Each encoded character is assigned a code value, or code point, that is unique to that character. This value is part of a Unicode character-mapping table whose values are controlled by the Unicode Consortium.

Code points appear as a U+ (or a backslash, \) followed by the hexadecimal value of the character, with the valid range of values from U+0000 to U+10FFFF$_{16}$. Combined characters, such as *Ä*, can be broken down into their individual components (*A* with an umlaut, in this case) and recomposed into their original state. The decomposition mapping for *A* is U+0041, and U+0308 for the umlaut. I will examine some Oracle functions in the next section that enable you to work with these code points.

A code unit is the byte size of the datatype that is used to store characters. The size of the code unit depends on the character set that is used. In certain circumstances, the code point is too large for a single code unit, so multiple code units are needed for a single code point.

Of course, users recognize characters, not code points and code units. The "word" \0053\0074\0065\0076\0065\006E doesn't mean a lot to average end users who recognize characters in their native languages. For one thing, the text actually displayed on the user's screen is called a *glyph* and is simply a representation of the underlying code point. Your computer may not have the required fonts or may be otherwise unable to render the characters on the screen. This does not mean, however, that Oracle has stored the code points incorrectly.

Unicode and Your Environment

Oracle supports all characters in the world, but does your environment? Unless you work with Unicode characters on a regular basis, there is a good chance that your system is not set up to support certain multibyte characters (is unable, that is, to render the proper glyphs). Operating system Unicode support does not guarantee that all applications on that operating system will work with all characters. The individual application vendors control their Unicode support. Even basic applications such as DOS have difficulty with certain characters if not properly configured.

If you require interaction with the Oracle database in a way that supports multibyte characters, but you do not want or need to adjust your operating system and application settings, consider configuring *Oracle Application Express* from Oracle. You can access the database using your browser, where Unicode encoding is easily configured. Oracle Application Express is free to install on top of any version of the Oracle database, and is actually preconfigured with Oracle Express Edition. *iSQL*Plus* is another option for Oracle9*i* Database and later.

Many web-based tools have the appropriate encoding scheme listed in their page headers so Unicode characters will display correctly by default, but if you do not see the correct characters, set the encoding in your browser as follows:

In Internet Explorer, select

> *View→Encoding→Auto-Select or Unicode (UTF-8).*

Using Firefox, select

> *View→Character Encoding→Unicode (UTF-8).*

Globalization Support Parameters

Oracle relies on Globalization Support (NLS) parameters for its default behavior. These settings are configured at the time of database creation and encompass everything from your character sets to default currency symbols. I will refer to parameters that can be modified for your session throughout this chapter. Table 25-3 shows the parameters, example values, and an explanation of each. You can find the values on your system from the NLS_SESSION_PARAMETERS view.

Table 25-3. NLS session parameters

Parameter	Description	Example
NLS_CALENDAR	Sets the default calendar for the database.	GREGORIAN
NLS_COMP	Works with NLS_SORT to define sort rules for characters. You must use a linguistic index when setting to ANSI.	BINARY
NLS_CURRENCY	Specifies the currency symbol and is based on the NLS_TERRITORY value unless explicitly overridden with another value.	$
NLS_DATE_FORMAT	The default format of the date only. It is derived from NLS_TERRITORY and can be overridden.	DD-MON-RR
NLS_DATE_LANGUAGE	Determines the spelling of the day and month for date-related functions.	AMERICAN
NLS_DUAL_CURRENCY	Helps support the euro and is derived from NLS_TERRITORY unless overridden. It is an alternate currency for a territory.	$
NLS_ISO_CURRENCY	The ISO currency symbol whose default is derived from NLS_TERRITORY. It can be overridden with any valid territory.	AMERICA
NLS_LANGUAGE	Sets the default language used within the database. It impacts everything from date formatting to server messages.	AMERICAN
NLS_LENGTH_SEMANTICS	Determines whether character or byte semantics are used.	BYTE
NLS_NCHAR_CONV_EXCP	Determines whether a character type conversion will report an error.	FALSE
NLS_NUMERIC_CHARACTERS	The default decimal character and group separator; derived from NLS_TERRITORY but can be overridden.	.,
NLS_SORT	Defines the character sort order for a given language.	BINARY
NLS_TERRITORY	Specifies the primary region supported by the database. This has a broad impact because many other NLS parameters depend on this value for their defaults.	AMERICA
NLS_TIMESTAMP_FORMAT	Default timestamp format for TO_TIMESTAMP and TO_CHAR functions.	DD-MON-RR HH.MI.SSXF F AM
NLS_TIMESTAMP_TZ_FORMAT	Sets the timestamp with time zone format for TO_CHAR and TO_TIMESTAMP_TZ.	DD-MON-RR HH.MI.SSXF F AM TZR
NLS_TIME_FORMAT	Complements the NLS_DATE_FORMAT mentioned earlier. Sets the default time format for the database.	HH.MI.SSXF F AM
NLS_TIME_TZ_FORMAT	Defines the time format including the time zone region or UTC offset.	HH.MI.SSXF F AM TZR

Unicode Functions

Oracle's Unicode PL/SQL support begins with some basic string functions. However, you will notice slight variations in Table 25-4 for some well-known functions: INSTR, LENGTH, and SUBSTR have a B, C, 2, or 4 appended to the end of the name indicating whether the function is byte-, character-, code unit–, or code point–based.

INSTR, LENGTH, and SUBSTR use the length semantics associated with the datatype of the column or variable. These base functions and the variations ending in *C* will often return the same value until you begin working with NCHAR or NVARCHAR values. Because your NLS_NCHAR_CHARACTERSET and NLS_CHARACTERSET can be different, INSTR, LENGTH, and SUBSTR can return different results (depending on the datatype) from their character counterparts.

Table 25-4. Unicode functions

Unicode function	Description
ASCIISTR(*string*)	Converts *string* to ASCII characters. When the character is Unicode, it formats as the standard Unicode format \xxxx.
COMPOSE(*string*)	Converts a decomposed string to its fully composed form.
DECOMPOSE(*string*, [*canonical* | *compatibility*])	Takes *string* as an argument and returns a Unicode string in its individual code points.
INSTRB(*string*, *substr*, *pos*, *occ*)	Returns the byte position of *substr* in *string* beginning at position *pos*. You can also specify the *occ* (occurrence) of *substr* if it appears more than once. The default of *pos* and *occ* are both 1 if not specified. *pos* is in bytes.
INSTRC(*string*, *substr*, *pos*, *occ*)	Similar to INSTRB except that it returns the character position of *substr* in *string* beginning at position *pos*, where *pos* is in characters.
INSTR2(*string*, *substr*, *pos*, *occ*)	Return position is based on UTF-16 code units.
INSTR4(*string*, *substr*, *pos*, *occ*)	Return position is based on UTF-16 code points.
LENGTHB(*string*)	Returns the number of bytes in *string*.
LENGTHC(*string*)	Returns the Unicode length of *string*. The length is in number of characters.
LENGTH2(*string*)	Length is based on UTF-16 code units.
LENGTH4(*string*)	Length is based on UTF-16 code points.
SUBSTRB(*string*, *n*, *m*)	Returns a portion of *string* beginning at position *n* for length *m*. *n* and *m* are in bytes.
SUBSTRC(*string*, *n*, *m*)	Returns a portion of *string* beginning at position *n* for length *m*. *n* and *m* are based on Unicode characters.
SUBSTR2(*string*, *n*, *m*)	*n* and *m* are in UTF-16 code units.
SUBSTR4(*string*, *n*, *m*)	*n* and *m* are in UTF-16 code points.
UNISTR	Converts *string* to an ASCII string representation of Unicode using backslash and hex digits.

Let's take a closer look at these functions.

ASCIISTR

ASCIISTR takes *string* as input and attempts to convert it to a string of ASCII characters. If *string* contains non-ASCII characters, it formats them as \xxxx. As you will see with the DECOMPOSE function described later, this formatting comes in very handy. The following example:

```
BEGIN
   DBMS_OUTPUT.put_line ('ASCII Character: ' || ASCIISTR ('A'));
   DBMS_OUTPUT.put_line ('Unicode Character: ' || ASCIISTR ('Ä'));
END;
```

returns this:

```
ASCII Character: A
Unicode Character: \00C4
```

COMPOSE

For some characters, there are multiple ways for code points to represent the same thing. This is a problem when you are comparing two values. For example, you can create an *Ä* using the single code point U+00C4, or with multiple code points U+0041 (the letter *A*) and U+0308 (the umlaut). U+00C4 is precomposed, while U+0041 and U+0308 are decomposed: On comparison, however, PL/SQL says these are not equal. That is, this query:

```
DECLARE
   v_precomposed   VARCHAR2 (20) := UNISTR ('\00C4');
   v_decomposed    VARCHAR2 (20) := UNISTR ('A\0308');
BEGIN
   IF v_precomposed = v_decomposed
   THEN
      DBMS_OUTPUT.put_line ('==EQUAL==');
   ELSE
      DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
   END IF;
END;
```

returns the following:

```
<>NOT EQUAL<>
```

Using the COMPOSE function, I can make the decomposed value equal to the precomposed value:

```
DECLARE
   v_precomposed   VARCHAR2 (20) := UNISTR ('\00C4');
   v_decomposed    VARCHAR2 (20) := COMPOSE (UNISTR ('A\0308'));
BEGIN
   IF v_precomposed = v_decomposed
```

```
      THEN
         DBMS_OUTPUT.put_line ('==EQUAL==');
      ELSE
         DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
      END IF;
   END;
```

This query returns the following:

```
==EQUAL==
```

DECOMPOSE

As you might have guessed, DECOMPOSE is the opposite of COMPOSE. DECOMPOSE takes something that is precomposed and breaks it down into separate code points or elements:

```
DECLARE
   v_precomposed   VARCHAR2 (20) := ASCIISTR (DECOMPOSE ('Ä'));
   v_decomposed    VARCHAR2 (20) := 'A\0308';
BEGIN
   IF v_precomposed = v_decomposed
   THEN
      DBMS_OUTPUT.put_line ('==EQUAL==');
   ELSE
      DBMS_OUTPUT.put_line ('<>NOT EQUAL<>');
   END IF;
END;
```

The results are as follows:

```
==EQUAL==
```

INSTR/INSTRB/INSTRC/INSTR2/INSTR4

All INSTR functions return the position of a substring within a string. The differences lie in how the position is determined:

INSTR
: Finds the position by character

INSTRB
: Returns the position in bytes

INSTRC
: Determines the position by Unicode character

INSTR2
: Uses code units to determine the position

INSTR4
: Returns the position by code point

To illustrate, let's use the publication table in the g11n schema:

```
DECLARE
   v_instr    NUMBER (2);
   v_instrb   NUMBER (2);
   v_instrc   NUMBER (2);
   v_instr2   NUMBER (2);
   v_instr4   NUMBER (2);
BEGIN
   SELECT INSTR (title, 'グ'),
 INSTRB (title, 'グ'),
 INSTRC (title, 'グ
'),
 INSTR2 (title, 'グ'),
 INSTR4 (title, 'グ
')
     INTO v_instr, v_instrb, v_instrc,
          v_instr2, v_instr4
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line ('INSTR of グ: ' || v_instr);
   DBMS_OUTPUT.put_line ('INSTRB of グ: ' || v_instrb);
   DBMS_OUTPUT.put_line ('INSTRC of グ: ' || v_instrc);
   DBMS_OUTPUT.put_line ('INSTR2 of グ: ' || v_instr2);
   DBMS_OUTPUT.put_line ('INSTR4 of グ: ' || v_instr4);
END;
/
```

The output is as follows:

```
INSTR of グ: 16
INSTRB of グ: 20
INSTRC of グ: 16
INSTR2 of グ: 16
INSTR4 of グ: 16
```

The position of character グ is different only for INSTRB in this case. One nice feature of INSTR2 and INSTR4 is that you can use them to search for code points that do not represent complete characters. Returning to our character *Ä*, it is possible to include the umlaut as the substring for which to search.

LENGTH/LENGTHB/LENGTHC/LENGTH2/LENGTH4

The LENGTH functions operate as follows:

LENGTH
Returns the length in characters of a string

LENGTHB
Returns the length in bytes of a string

LENGTHC
: Returns the length in Unicode characters of a string

LENGTH2
: Gives the number of code units in a string

LENGTH4
: Gives the number of code points in a string

The LENGTH function matches the LENGTHC function when characters are precomposed:

```
DECLARE
   v_length    NUMBER (2);
   v_lengthb   NUMBER (2);
   v_lengthc   NUMBER (2);
   v_length2   NUMBER (2);
   v_length4   NUMBER (2);
BEGIN
   SELECT LENGTH (title), LENGTHB (title), lengthc (title), length2 (title),
          length4 (title)
     INTO v_length, v_lengthb, v_lengthc, v_length2,
          v_length4
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line ('LENGTH of string: ' || v_length);
   DBMS_OUTPUT.put_line ('LENGTHB of string: ' || v_lengthb);
   DBMS_OUTPUT.put_line ('LENGTHC of string: ' || v_lengthc);
   DBMS_OUTPUT.put_line ('LENGTH2 of string: ' || v_length2);
   DBMS_OUTPUT.put_line ('LENGTH4 of string: ' || v_length4);
END;
```

This gives the following result:

```
LENGTH of string: 28
LENGTHB of string: 52
LENGTHC of string: 28
LENGTH2 of string: 28
LENGTH4 of string: 28
```

The only difference in this case is with the LENGTHB function. As expected, LENGTH and LENGTHC returned the same result. My expectation changes when I'm working with decomposed characters, however. Note the following example:

```
DECLARE
   v_length   NUMBER (2);
BEGIN
   SELECT LENGTH (UNISTR ('A\0308'))
     INTO v_length
     FROM DUAL;

   DBMS_OUTPUT.put_line ('Decomposed string size using LENGTH: ' || v_length);
   SELECT lengthc (UNISTR ('A\0308'))
     INTO v_length
     FROM DUAL;

   DBMS_OUTPUT.put_line ('Decomposed string size using LENGTHC: ' || v_length);
END;
```

The length is returned as follows:

```
Decomposed string size using LENGTH: 2
Decomposed string size using LENGTHC: 1
```

In this case, LENGTH still returns the number of characters, but it sees the *A* as separate from the umlaut. LENGTHC returns the length of the string in Unicode characters, so it sees only one character.

SUBSTR/SUBSTRB/SUBSTRC/SUBSTR2/SUBSTR4

The different versions of SUBSTR follow the same pattern as INSTR and LENGTH. SUBSTR returns a portion of a string of a specified length, beginning at a given position. The variations of the function operate as follows:

SUBSTR
: Determines position and length by character

SUBSTRB
: Determines position and length in bytes

SUBSTRC
: Determines position and length using Unicode characters

SUBSTR2
: Uses code units

SUBSTR4
: Uses code points

The following example illustrates the use of these functions:

```
DECLARE
   v_substr    VARCHAR2 (20);
   v_substrb   VARCHAR2 (20);
   v_substrc   VARCHAR2 (20);
   v_substr2   VARCHAR2 (20);
   v_substr4   VARCHAR2 (20);
BEGIN
   SELECT SUBSTR (title, 13, 4), SUBSTRB (title, 13, 4),
          substrc (title, 13, 4), substr2 (title, 13, 4),
          substr4 (title, 13, 4)
     INTO v_substr, v_substrb,
          v_substrc, v_substr2,
          v_substr4
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line ('SUBSTR of string: ' || v_substr);
   DBMS_OUTPUT.put_line ('SUBSTRB of string: ' || v_substrb);
   DBMS_OUTPUT.put_line ('SUBSTRC of string: ' || v_substrc);
   DBMS_OUTPUT.put_line ('SUBSTR2 of string: ' || v_substr2);
   DBMS_OUTPUT.put_line ('SUBSTR4 of string: ' || v_substr4);
END;
```

Notice the difference between SUBSTRB and the other functions in the output from the script:

```
SUBSTR of string: L プログ
SUBSTRB of string: L プ
SUBSTRC of string: L プログ
SUBSTR2 of string: L プログ
SUBSTR4 of string: L プログ
```

UNISTR

UNISTR takes a string and converts it to Unicode. I used this function in a few earlier examples to display the Unicode character corresponding to a decomposed string. In the section "Character Encoding" on page 1102, I used a string of code points as an example when discussing glyphs. I can use UNISTR to make sense of all this:

```
DECLARE
   v_string   VARCHAR2 (20);
BEGIN
   SELECT UNISTR ('\0053\0074\0065\0076\0065\006E')
     INTO v_string
     FROM DUAL;

   DBMS_OUTPUT.put_line (v_string);
END;
```

The output is as follows:

```
Steven
```

Character Semantics

Undoubtedly, one of the first issues you will run into when localizing your application is support for multibyte characters. When you pass your first Japanese characters to a VARCHAR2 variable and experience an ORA-6502 error, you will likely spend an hour debugging your procedure that "should work."

At some point, you may realize that every declaration of every character variable or character column in your application will have to be changed to accommodate the multibyte character set. You will then, if you are anything like me, consider for a moment changing careers. Don't give up! Once you work through the initial challenges, you will be in a very strong position to guide application implementations in the future.

Consider the following example:

```
DECLARE
   v_title   VARCHAR2 (30);
BEGIN
   SELECT title
     INTO v_title
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line (v_title);
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

It returns the following exception:

```
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
```

It failed because the precision of 30 is in bytes, not in characters. A number of Asian character sets have up to 3 bytes per character, so it's possible that a variable with a precision of 2 will actually not support even a single character in your chosen character set!

Using the LENGTHB function, I can determine the actual size of the string:

```
DECLARE
   v_length_in_bytes   NUMBER (2);
BEGIN
   SELECT LENGTHB (title)
     INTO v_length_in_bytes
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line ('String size in bytes: ' || v_length_in_bytes);
END;
```

This returns the following result:

```
String size in bytes: 52
```

Prior to Oracle9*i* Database, we were somewhat limited in what we could do. The approach I most frequently used in Oracle8*i* Database was to simply use the maximum number of characters expected and multiply that by 3:

```
DECLARE
   v_title   VARCHAR2 (90);
BEGIN
   SELECT title
```

```
     INTO v_title
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line (v_title);
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

If you are using a display that can render the proper glyph, the following result is returned:

```
Oracle PL/SQL プログラミング 基礎編 第 3 版
```

This workaround does the job, but it is clumsy. Using byte semantics and simply multiplying the number of expected characters by 3 causes some undesired behaviors in your application:

- Many other database vendors use character rather than byte semantics by default, so porting applications to multiple databases becomes cumbersome.
- In cases where characters do not take the full 3 bytes, it is possible for the variable or column to store more than the expected number of characters.
- The padding that Oracle automatically applies to CHAR datatypes means that the full 90 bytes are used regardless of whether they are needed.

Character semantics were first introduced in Oracle9*i* Database. It is possible to declare a variable with precision in either bytes or characters. The following example is the same one that failed earlier—with one exception. Look at the declaration of the variable to see how I invoke character semantics:

```
DECLARE
   v_title   VARCHAR2 (30 CHAR);
BEGIN
   SELECT title
     INTO v_title
     FROM publication
    WHERE publication_id = 2;

   DBMS_OUTPUT.put_line (v_title);
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

This returns the following complete string:

```
Oracle PL/SQL プログラミング 基礎編 第 3 版
```

This method still requires a change for every character variable or column declaration in your application. An easier solution is to change from byte semantics to character semantics for your entire database. To make this change, simply set NLS_LENGTH_SEMANTICS to CHAR. You can find your current setting by running:

```
SELECT parameter, VALUE
  FROM nls_session_parameters
 WHERE parameter = 'NLS_LENGTH_SEMANTICS'
```

Assuming the database was created using byte semantics, the following is returned:

```
PARAMETER                 VALUE
------------------------- ----------
NLS_LENGTH_SEMANTICS      BYTE
```

Also check the V$PARAMETER view:

```
SELECT NAME, VALUE
  FROM v$parameter
 WHERE NAME = 'nls_length_semantics'
```

This query returns the following:

```
NAME                      VALUE
------------------------- ----------
nls_length_semantics      BYTE
```

To switch to character semantics, modify your system NLS_LENGTH_SEMANTICS setting using the ALTER SYSTEM command:

```
ALTER SYSTEM SET NLS_LENGTH_SEMANTICS = CHAR
```

You can also modify this parameter for a session with the ALTER SESSION command:

```
ALTER SESSION SET NLS_LENGTH_SEMANTICS = CHAR
```

With this approach, modifying an existing application becomes a snap; now all existing declarations are automatically based on number of characters rather than bytes. After setting the system to use character semantics, you can see the change in the data dictionary:

```
SELECT parameter, value
  FROM nls_session_parameters
 WHERE parameter = 'NLS_LENGTH_SEMANTICS'
```

The following is returned:

```
PARAMETER                 VALUE
------------------------- ----------
NLS_LENGTH_SEMANTICS      CHAR
```

Returning to the prior example, you can see that character semantics are used without specifying CHAR in the declaration:

```
DECLARE
   v_title   VARCHAR2 (30);
BEGIN
   SELECT title
     INTO v_title
     FROM publication
     WHERE publication_id = 2;

   DBMS_OUTPUT.put_line (v_title);
EXCEPTION
   WHEN OTHERS
   THEN
      DBMS_OUTPUT.put_line (DBMS_UTILITY.format_error_stack);
END;
```

The following is returned:

```
Oracle PL/SQL プログラミング 基礎編 第 3 版
```

Note that the maximum number of bytes allowed is not adjusted in any way with character semantics. While switching to character semantics will allow 1,000 3-byte characters to go into a VARCHAR2(1000) without modification, you will not be able to put 32,767 3-byte characters in a VARCHAR2(32767). The VARCHAR2 variable limit is still set at 32,767 bytes and the VARCHAR2 column maximum is still 4,000 bytes.

Include the use of character semantics in your initial application design and make your life infinitely easier. Unless you have an overwhelming need to use byte semantics in a portion of your application, set the parameter NLS_LENGTH_SEMANTICS = CHAR to make character semantics the default. If you change your NLS_LENGTH_SEMANTICS setting for an existing application, remember to recompile all objects so the change will take effect. *This includes rerunning the catproc.sql script to re-create all supplied packages too!*

String Sort Order

Oracle provides advanced linguistic sort capabilities that extend far beyond the basic A–Z sorting you get with an ORDER BY clause. The complexities found in international character sets do not lend themselves to simple alphabetic sort, or collation, rules. Chinese, for example, includes approximately 70,000 characters (although many are not used regularly)—not exactly something you can easily put into song like the ABCs! It's also not something that can be defined by simple sort rules.

String sort order is an obvious programming problem that is often overlooked in globalization until a product makes its way to the test team. In many languages, ordering the names of employees, cities of operation, or customers is much more complicated than "A comes before B." Consider the following factors:

- Some European characters include accents that change the meaning of the base letter. The letter *a* is different from *ä*. Which letter should come first in an ORDER BY?
- Each locale may have its own sort rules, so a multilingual application must be able to support different sort rules based on the text. Even regions that use the same alphabet may still have different sort rules.

Oracle provides three types of sorts: binary, monolingual, and multilingual.

The Unicode Consortium makes its collation algorithm public, so we can compare the output from our queries for these three types of sorts with the expected results shown in the collation charts on the Unicode website.

Binary Sort

The binary sort is based on the character encoding scheme and the values associated with each character. It is very fast, and is especially useful if you know that all of your data is stored in uppercase. The binary sort is most useful for ASCII characters, sorting the English alphabet, but even then you may find some undesired results. ASCII encoding, for example, orders uppercase letters before their lowercase representations.

The following example from the g11n sample schema shows the results of a binary sort of German cities:

```
  SELECT city
    FROM store_location
   WHERE country <> 'JP'
ORDER BY city;
```

The ordered list of results is as follows:

```
CITY
------------------------------------
Abdêra
Asselfingen
Astert
Auufer
Außernzell
Aßlar
Boßdorf
Bösleben
Bötersen
Cremlingen
Creuzburg
Creußen
Oberahr
```

```
Zudar
Zühlen
Ängelholm
...lsen
```

Note the order of the cities in the list. Ängelholm is ordered after Zühlen. Character codes are sorted in ascending order, providing the A–Z ordering you see. These anomalies result from the inclusion of characters outside the English alphabet, here being treated as special characters.

Monolingual Sort

Most European languages will benefit from monolingual sort capabilities within Oracle. Rather than using basic codes associated with the character encoding scheme like the binary sort, two values are used to determine the relative position of a character in a monolingual sort. Each character has a major value, related to the base character, and a minor value, based on case and diacritic differences. If sort order can be determined by a difference in major value, the ordering is complete. Should there be a tie in major value, the minor value is used. This way, characters such as *ö* can be ordered relative to the character *o* accurately.

To see the impact this has on the ordering of these additional characters, let's return to the prior example and modify the session to use the German monolingual sort:

```
ALTER SESSION SET NLS_SORT = german;
```

Upon confirmation that the session has been altered, I run the following query:

```
  SELECT city
    FROM store_location
   WHERE country <> 'JP'
ORDER BY city;
```

Notice that the order is different now that NLS_SORT is set to german:

```
CITY
------------------------------------
Abdêra
Ängelholm
Aßlar
Asselfingen
Astert
Außernzell
Auufer
Boßdorf
Bösleben
Bötersen
Cremlingen
Creußen
Creuzburg
Oberahr
```

```
...lsen
Zudar
Zühlen
```

This is much better! The treatment of non-English characters is now in line with the expected German order of characters. By the way, if you do not want to (or cannot) alter your session's NLS settings, you can use the NLSSORT function and the NLS_SORT parameter as part of your query. The following function demonstrates:

```
FUNCTION city_order_by_func (v_order_by IN VARCHAR2)
   RETURN sys_refcursor
IS
   v_city   sys_refcursor;
BEGIN
   OPEN v_city
    FOR
       SELECT city
         FROM store_location
     ORDER BY NLSSORT (city, 'NLS_SORT=' || v_order_by);

   RETURN v_city;
END city_order_by_func;
```

As shown here, the NLSSORT function and the NLS_SORT parameter provide quick ways to change the results of an ORDER BY clause. For this function, which is used in the remaining examples, the NLS_SORT parameter is taken as input. Table 25-5 lists some of the available NLS_SORT parameter values in recent versions of the Oracle Database.

Table 25-5. Monolingual NLS_SORT parameter values

arabic	xcatalan	japanese
arabic_abj_sort	german	polish
arabic_match	xgerman	punctuation
arabic_abj_match	german_din	xpunctuation
azerbaijani	xgerman_din	romanian
xazerbaijani	hungarian	russian
bengali	xhungarian	spanish
bulgarian	icelandic	xspanish
canadian french	indonesian	west_european
catalan	italian	xwest_european

The list of parameters in this table includes some values prefixed with an *x*. These extended sort parameters allow for special cases in a language. In my cities example, some names have the character ß. This *sharp s* in German can be treated as *ss* for the purposes of the sort. I tried a sort using NLS_SORT = german earlier. Let's try xgerman to see the difference:

```
VARIABLE v_city_order REFCURSOR
CALL city_order_by_func('xgerman') INTO :v_city_order;
PRINT v_city_order
```

This displays the following:

```
CITY
----------------------------------
...
Abdêra
Ängelholm
Asselfingen
Aßlar
Astert
Außernzell
Auufer
...
```

Using xgerman rather than german, the word Aßlar drops to fourth in the list instead of third.

Multilingual Sort

Monolingual sort, as you might guess from the use of "mono" in its name, has a major drawback. It can operate on only one language at a time, based on the NLS_SORT parameter setting. Oracle also provides *multilingual* sort capabilities that allow you to sort for multiple locales.

The multilingual sort, based on the ISO 14651 standard, supports more than 1.1 million characters in a single sort. Not only does Oracle's multilingual support cover characters defined as part of the Unicode standard, but it can also support supplementary characters.

Whereas binary sorts are determined by character encoding scheme codes and monolingual sorts are developed in two stages, multilingual sorts use a three-step approach to determine the order of characters:

1. The first level, or primary level, separates base characters.
2. The secondary level distinguishes base characters from diacritics that modify the base characters.
3. Finally, the tertiary level separates by case.

For Asian languages, characters are also differentiated by number of strokes, PinYin, or radicals.

NLSSORT and NLS_SORT are still used for multilingual sorts, but the parameters change. GENERIC_M works well for most Western languages, and provides the base for the remaining list of values. Table 25-6 lists the NLS_SORT parameter values available for multilingual sorts.

Table 25-6. Multilingual NLS_SORT parameter values

generic_m			
canadian_m	japanese_m	schinese_pinyin_m	tchinese_radical_m
danish_m	korean_m	schinese_radical_m	tchinese_stroke_m
french_m	schinese_stroke_m	spanish_m	thai_m

To demonstrate the multilingual sort functionality, I can modify the call to use the generic_m value:

```
VARIABLE v_city_order REFCURSOR
CALL city_order_by_func('generic_m') INTO :v_city_order;
PRINT v_city_order
```

This returns the following ordered list of cities:

```
CITY
------------------------------------
Abdêra
Ängelholm
Asselfingen
Aßlar
Astert
..
Zudar
Zühlen
尼崎市
旭川市
足立区
青森市
```

Multilingual Information Retrieval

Application developers who work on catalogs, digital libraries, and knowledge repositories are no doubt familiar with information retrieval, or IR. An IR application takes in user-supplied criteria and searches for the items or documents that best match the *intent* of the user. This is one of the major ways that IR differs from standard SQL queries, which either do or do not find matches for the query criteria. Good IR systems can help determine what documents are about and return those documents that are most relevant to the search, even if they don't match the search exactly.

Perhaps the most challenging task in IR is to support indexing and querying in multiple languages. English, for example, is a single-byte language that uses whitespace to separate words. Retrieval of information is substantially different when working with Japanese, which is a multibyte character set that does *not* use whitespace as delimiters.

Oracle Text, an option available in both the Oracle Enterprise Edition and the Oracle Standard Edition, provides full-text IR capabilities. Because Oracle Text uses SQL for index creation, search, and maintenance operations, it works very well in PL/SQL-based applications.

Called ConText and *inter*Media in prior releases, Oracle Text really came of age as an information retrieval solution with Oracle9*i* Database. With Oracle Text:

- All NLS character sets are supported.
- Searching across documents in Western languages—as well as in Korean, Japanese, and Traditional and Simplified Chinese—is possible.
- Unique characteristics of each language are accommodated.
- Searches are case insensitive by default.
- Cross-language search is supported.

Before a PL/SQL application can be written that searches a data source, Oracle Text indexes must be created. As part of the g11n schema, I created an Oracle Text index on the publication.short_description column. To support multiple languages, I provided individual language preferences, as well as a MULTI_LEXER preference that makes it possible to search across multiple languages with a single query.

Oracle Text Indexes

There are four index types available with Oracle Text. The first, and most commonly used, is the CONTEXT index. The CONTEXT index can index any character or LOB column, including BFILEs. It uses a filter to extract text from different document types. The filter that is shipped with the data server can filter more than 350 different document types, including Word documents, PDFs, and XML documents.

Once the text is extracted from the document, it is broken into *tokens*, or individual terms and phrases, by a LEXER. Language-specific LEXERs are available where required. A MULTI_LEXER actually uses language-specific LEXERs (defined as SUB_LEXERs) to extract the tokens from a multilingual data source. The tokens are stored in Oracle Text index tables and used during search operations to point to relevant documents. To see the tokens created in this chapter's examples, run the following:

```
SELECT token_text
  FROM dr$g11n_index$i
```

The result contains English, German, and Japanese tokens.

The other three Oracle Text index types are the CTXCAT, CTXRULE, and CTXXPATH indexes. For additional information regarding their structure, check out Oracle's *Text Application Developer's Guide* and *Text Reference*.

You can use the TEXT_SEARCH_FUNC function that is part of the g11n schema to test some of the multilingual features:

```
FUNCTION text_search_func (v_keyword IN VARCHAR2)
   RETURN sys_refcursor
IS
   v_title   sys_refcursor;
BEGIN
   OPEN v_title
    FOR
       SELECT   title, LANGUAGE, score (1)
         FROM publication
         WHERE contains (short_description, v_keyword, 1) > 0
      ORDER BY score (1) DESC;

   RETURN v_title;
END text_search_func;
```

A call to this function, passing "pl" as the keyword:

```
variable x refcursor;
call text_search_func('pl') into :x;
print x;
```

returns the following result:

```
TITLE                                              LANGUAGE  SCORE(1)
-----------------------------------                --------  --------
Oracle PL/SQL プログラミング 基礎編 第 3 版                 JA        18
Oracle PL/SQL Programming, 3rd Edition            EN        13
Oracle PL/SQL Programmierung, 2. Auflage          DE        9
```

You find this reference in all three languages because "pl" is common among them. Note that I searched on a lowercase "pl", but the "PL" in the record is uppercase. My search is case insensitive by default even though no UPPER function was used.

It may be that some languages should be case sensitive while others should not be. Case sensitivity can be specified on a per-language basis when you provide your language preferences. Simply add a mixed_case attribute with a value of yes; the tokens will be created in mixed case, just as they are stored in your document or column, but only for the language identified in that preference.

Oracle Database 11*g* made multilingual IR easier with the introduction of the AUTO_LEXER. Although it has fewer language-specific features than the MULTI_LEXER, it provides a nearly effortless method of implementation. Instead of relying on a language column, the AUTO_LEXER identifies the text based on the code point.

Oracle also supports the WORLD_LEXER. It is not as full-featured as the MULTI_LEXER, and does not have language-specific features like the AUTO_LEXER. It is very easy to configure, however.

With the WORLD_LEXER, the text is broken into tokens based on the category in which it falls. Text in languages that fall into categories such as Arabic and Latin, where tokens are separated by whitespace, can easily be divided on that basis. Asian characters are more complex because they aren't whitespace delimited, so they are broken into overlapping tokens of two characters at a time. For example, the three-character string of 尼崎市 is broken into two tokens, 尼崎 and 崎市.

Oracle Text provides additional features as well, depending on the language. For details including language-specific features and restrictions, see the Oracle Text documentation provided on the OTN website.

IR and PL/SQL

I have designed and implemented some extremely large and complex record management systems and digital libraries. In my experience, nothing beats PL/SQL for search and maintenance operations with Oracle Text. PL/SQL's tight integration with the database server, and its improved performance over the last few releases, makes stored PL/SQL program units the ideal choice for these types of applications.

This is even more evident when you're working with multiple languages. The shared SQL and PL/SQL parser means that there is consistent handling of characters and character semantics, regardless of the language being indexed and searched.

One of the first projects most Oracle Text programmers undertake is to find a way to format strings for search. The following example creates a function that formats search strings for Oracle Text:

```
/* File on web: g11n.sql */
FUNCTION format_string (p_search IN VARCHAR2)
   RETURN VARCHAR2
AS
-- Define an associative array
   TYPE token_table IS TABLE OF VARCHAR2 (500 CHAR)
      INDEX BY PLS_INTEGER;

-- Define an associative array variable
   v_token_array           token_table;
   v_temp_search_string    VARCHAR2 (500 CHAR);
   v_final_search_string   VARCHAR2 (500 CHAR);
   v_count                 PLS_INTEGER         := 0;
   v_token_count           PLS_INTEGER         := 0;
BEGIN
   v_temp_search_string := TRIM (UPPER (p_search));
   -- Find the max number of tokens
   v_token_count :=
        lengthc (v_temp_search_string)
      - lengthc (REPLACE (v_temp_search_string, ' ', ''))
      + 1;
```

```
-- Populate the associative array
FOR y IN 1 .. v_token_count
LOOP
   v_count := v_count + 1;
   v_token_array (y) :=
         regexp_substr (v_temp_search_string, '[^[:space:]]+', 1, v_count);
   -- Handle reserved words
   v_token_array (y) := TRIM (v_token_array (y));

   IF v_token_array (y) IN ('ABOUT', 'WITHIN')
   THEN
      v_token_array (y) := '{' || v_token_array (y) || '}';
   END IF;
END LOOP;

v_count := 0;

FOR y IN v_token_array.FIRST .. v_token_array.LAST
LOOP
   v_count := v_count + 1;

   -- First token processed
   IF (    (v_token_array.LAST = v_count OR v_count = 1)
       AND v_token_array (y) IN ('AND', '&', 'OR', '|')
      )
   THEN
      v_final_search_string := v_final_search_string;
   ELSIF (v_count <> 1)
   THEN
      -- Separate by a comma unless separator already present
      IF    v_token_array (y) IN ('AND', '&', 'OR', '|')
         OR v_token_array (y - 1) IN ('AND', '&', 'OR', '|')
      THEN
         v_final_search_string :=
                         v_final_search_string || ' ' || v_token_array (y);
      ELSE
         v_final_search_string :=
                        v_final_search_string || ', ' || v_token_array (y);
      END IF;
   ELSE
      v_final_search_string := v_token_array (y);
   END IF;
END LOOP;

-- Escape special characters in the final string
v_final_search_string :=
   TRIM (REPLACE (REPLACE (v_final_search_string,
                           '&',
                           ' & '
                          ),
                  ';',
                  ' ; '
```

```
                      )
             );
      RETURN (v_final_search_string);
   END format_string;
```

This is designed to break *terms*, or tokens, from the string using the spaces between the characters. It uses character semantics for variable declarations, including the declaration of the associative array.

To test this with an English string, I run this SELECT:

```
SELECT format_string('oracle PL/SQL') AS "Formatted String"
  FROM dual
```

This returns the following result:

```
Formatted String
-----------------
ORACLE, PL/SQL
```

The FORMAT_STRING function separates terms with a comma by default, so an exact match is not required. A string of characters that is not whitespace-delimited will look exactly the way it was entered. The following example illustrates this using a mix of English and Japanese characters:

```
SELECT format_string('Oracle PL/SQL プログラミング 基礎編 第 3 版') AS
"Formatted String" FROM dual;
```

Passing this mixed character string to the FORMAT_STRING function returns the following result:

```
Formatted String
-----------------
ORACLE, PL/SQL プログラミング, 基礎編, 第 3 版
```

Where spaces delimit terms in the text, a comma is added regardless of the language.

The following CONTAINS search uses the FORMAT_STRING function:

```
SELECT score (1) "Rank", title
  FROM publication
  WHERE contains (short_description, format_string('プログラム'), 1) > 0;
```

This returns the following:

```
      Rank     TITLE
------------   ------------
          12   Oracle SQL*Plus デスクトップリファレンス
```

Using PL/SQL and Oracle Text, it is possible to index and perform full-text searches on data regardless of character set or language.

Date/Time

My globalization discussion thus far has been focused on strings. Date/time issues, however, can be every bit as troublesome for localizing an application. Users may be on the other side of the world from your database and web server, but they still require accurate information relating to their time zone, and the format of the date and time must be in a recognized structure.

Consider the following issues related to date/time:

- There are different time zones around the world.
- Daylight saving time exists for some regions, but not for others.
- Certain locales use different calendars.
- Date/time formatting is not consistent throughout the world.

Timestamp Datatypes

Until Oracle9*i* Database, working with dates and times was fairly straightforward. You had the DATE type and the TO_DATE function. The limited functionality of the DATE type made application development of global applications somewhat tedious, though. All time zone adjustments involved manual calculations. Sadly, if your application has to work with Oracle8*i* Database or earlier versions, I'm afraid you are still stuck with this as your only option.

Those of us working with Oracle9*i* Database and later, however, benefit greatly from the TIMESTAMP and INTERVAL datatypes discussed in detail in Chapter 10. If you have not read that chapter yet, I'll provide a quick overview here, but I do recommend that you go back and read that chapter to obtain a thorough understanding of the topic.

Let's take a look at an example of the TIMESTAMP, TIMESTAMP WITH TIME ZONE, and TIMESTAMP WITH LOCAL TIME ZONE datatypes in action:

```
DECLARE
   v_date_timestamp       TIMESTAMP ( 3 )                      := SYSDATE;
   v_date_timestamp_tz    TIMESTAMP ( 3 ) WITH TIME ZONE       := SYSDATE;
   v_date_timestamp_ltz   TIMESTAMP ( 3 ) WITH LOCAL TIME ZONE := SYSDATE;
BEGIN
   DBMS_OUTPUT.put_line ('TIMESTAMP:  ' || v_date_timestamp);
   DBMS_OUTPUT.put_line ('TIMESTAMP WITH TIME ZONE:  ' || v_date_timestamp_tz);
   DBMS_OUTPUT.put_line (   'TIMESTAMP WITH LOCAL TIME ZONE:  '
                         || v_date_timestamp_ltz
                        );
END;
```

The following dates and times are returned:

```
TIMESTAMP:                     08-JAN-05 07.28.39.000 PM
TIMESTAMP WITH TIME ZONE:      08-JAN-05 07.28.39.000 PM -07:00
TIMESTAMP WITH LOCAL TIME ZONE: 08-JAN-05 07.28.39.000 PM
```

TIMESTAMP and TIMESTAMP WITH LOCAL TIMESTAMP are identical because the database time is in the same locale as my session. The value for TIMESTAMP WITH TIMEZONE shows that I am in the Mountain time zone. If I were in accessing my Colorado database via a session in California, the result would be slightly different:

```
TIMESTAMP:                     08-JAN-05 07.28.39.000 PM
TIMESTAMP WITH TIME ZONE:      08-JAN-05 07.28.39.000 PM -07:00
TIMESTAMP WITH LOCAL TIME ZONE: 08-JAN-05 06.28.39.000 PM
```

The value for TIMESTAMP WITH LOCAL TIMEZONE used the time zone of my session, which is now Pacific, or –08:00, and automatically converted the value.

Date/Time Formatting

One localization challenge we face is related to date and time formatting. Japan, for example, may prefer the format *yyyy/MM/dd hh:mi:ssxff AM*, while in the United States you would expect to see *dd-MON-yyyy hh:mi:ssxff AM*.

A common way to handle this situation is to include a list of format masks in a locale table that maps to the user. When the user logs in, his assigned locale maps to the correct date/time format for his region.

The g11n schema has a USERS table and a LOCALE table, joined by a locale_id. Let's take a look at some examples using date/time functions (discussed in detail in Chapter 10) and the format masks provided in the g11n.locale table.

The registration_date column in the table uses the TIMESTAMP WITH TIME ZONE datatype. Using the TO_CHAR function and passing the format mask for each user's locale displays the date in the correct format:

```
/* File on web: g11n.sql */
FUNCTION date_format_func
   RETURN sys_refcursor
IS
   v_date   sys_refcursor;
BEGIN
   OPEN v_date
    FOR
       SELECT locale.locale_desc "Locale Description",
              TO_CHAR (users.registration_date,
                       locale.DATE_FORMAT
                      ) "Registration Date"
         FROM users, locale
        WHERE users.locale_id = locale.locale_id;

   RETURN v_date;
END date_format_func;
```

To execute it, I do the following:

```
variable v_format refcursor
CALL date_format_func() INTO :v_format;
PRINT v_format
```

This prints the following result set:

```
Locale Description          Registration Date
-----------------------     ------------------
English                     01-JAN-2005 11:34:21.000000 AM US/MOUNTAIN
Japanese                    2005/01/01 11:34:21.000000 AM JAPAN
German                      01 January 05 11:34:21.000000 AM EUROPE/WARSAW
```

The three locales have different date format masks assigned. Using this method allows each user to see an appropriate date format for her locale based on her profile. If I now add NLS_DATE_FORMAT, the dates and times will use the appropriate locale language. I have this mapped in my tables to ensure that each locale is displayed correctly:

```
/* File on web: g11n.sql */
FUNCTION date_format_lang_func
   RETURN sys_refcursor
IS
   v_date   sys_refcursor;
BEGIN
   OPEN v_date
    FOR
       SELECT locale.locale_desc "Locale Description",
              TO_CHAR (users.registration_date,
                       locale.DATE_FORMAT,
                       'NLS_DATE_LANGUAGE= ' || locale_desc
                      ) "Registration Date"
         FROM users, locale
        WHERE users.locale_id = locale.locale_id;

   RETURN v_date;
END date_format_lang_func;
```

I execute the function as follows:

```
variable v_format refcursor
CALL date_format_lang_func() INTO :v_format;
PRINT v_format
```

This prints the following:

```
Locale Description          Registration Date
-----------------------     ------------------
English                     01-JAN-2005 11:34:21.000000 AM US/MOUNTAIN
Japanese                    2005/01/01 11:34:21.000000 午前  JAPAN
German                      01 Januar 05 11:34:21.000000 AM EUROPE/WARSAW
```

The same data is stored in the USERS table, but the time is displayed in locale-specific format. I can modify the function in the same way to use the time zone and timestamp functions to distinguish between time zones for various locales. NLS_DATE_LANGUAGE is customized for each territory, so AM is in Japanese for the Japanese locale, and the month for the German locale is displayed in German.

I can extend my function to include the session time zone either by converting the value to the TIMESTAMP WITH TIME ZONE datatype, or by converting the value to my session's local time zone with TIMESTAMP WITH LOCAL TIME ZONE. I do this with the CAST function (described in Chapter 7), which will change the datatype of the value stored in my table:

```
/* File on web: g11n.sql */
FUNCTION date_ltz_lang_func
   RETURN sys_refcursor
IS
   v_date   sys_refcursor;
BEGIN
   OPEN v_date
    FOR
       SELECT locale.locale_desc,
              TO_CHAR
                 (CAST
                     (users.registration_date AS TIMESTAMP WITH LOCAL TIME ZONE
                     ),
                  locale.DATE_FORMAT,
                  'NLS_DATE_LANGUAGE= ' || locale_desc
                 ) "Registration Date"
         FROM users, locale
        WHERE users.locale_id = locale.locale_id;

   RETURN v_date;
END date_ltz_lang_func;
```

I execute the function by doing the following:

```
variable v_format refcursor
CALL date_ltz_lang_func() INTO :v_format;
PRINT v_format
```

The registration dates are returned as follows:

```
Locale Description           Registration Date
-----------------------      ------------------
English                      01-JAN-2005 11:34:21.000000 AM -07:00
Japanese                     2004/12/31 07:34:21.000000 午後 -07:00
German                       01 Januar 05 03:34:21.000000 AM -07:00
```

There is a lot going on here:

- Date/time language is converted to the locale-specific terms.
- Formatting is locale specific.

- I use CAST to convert the values stored as TIMESTAMP WITH TIMEZONE to TIMESTAMP WITH LOCAL TIMEZONE.
- The displayed time is relative to my session's time zone, which is US/Mountain in this example, or –07:00.

Many of my examples thus far have shown the time zone as a UTC offset. This is not necessarily the easiest display for a user to understand. Oracle maintains a list of region names and abbreviations that you can substitute by modifying the format mask. In fact, I inserted the three records I have been working with using these region names rather than the UTC offset. For a complete list of time zones, query the V$TIME-ZONE_NAMES view. Examine the INSERT statements into the USERS table in the g11n schema for more examples using region names.

I want to discuss one more NLS parameter related to date/time before we move on. I insert my times into the table using the Gregorian calendar, which is the value for NLS_CALENDAR on my test system. Not all locales use the same calendar, however, and no amount of formatting will adjust the base calendar. With NLS_CALENDAR, I can change my default calendar from Gregorian to a number of other seeded calendars —Japanese Imperial, for example. A simple SELECT of SYSDATE after I set the session results in the following:

```
ALTER SESSION SET NLS_CALENDAR = 'JAPANESE IMPERIAL';
ALTER SESSION SET NLS_DATE_FORMAT = 'E RR-MM-DD';
```

After altering the session, I run the following SELECT:

```
SELECT sysdate
  FROM dual;
```

The SELECT shows the modified SYSDATE:

```
SYSDATE
---------
H 17-02-08
```

Default values are controlled by your NLS settings. If you have a primary locale you are working with, you may find that setting your NLS parameters for your database is a much easier approach than explicitly stating them in your application. For applications in which these settings need to be dynamic, however, I recommend that you include NLS settings as part of your user/locale settings and store them with your application. When you set a user's profile correctly, your code will function in any locale.

Currency Conversion

A discussion of globalization and localization would not be complete without addressing currency conversion issues. The most common approach to the conversion from dollars to yen, for example, is to use a rate table that tracks conversion rates between monetary units. But how does an application know how to display the resulting number? Consider the following:

- Are decimals and commas appropriate, and where should they be placed?
- Which symbol is used for each currency ($ for dollar, € for euro, etc.)?
- Which ISO currency symbol should be displayed (USD, for example)?

Each publication in the g11n schema has a price and is associated with a locale. I can use the TO_CHAR function to format each string, but what about displaying the correct currency? I can use the NLS_CURRENCY parameter to format my prices correctly as follows:

```
/* File on web: g11n.sql */
FUNCTION currency_conv_func
   RETURN sys_refcursor
IS
   v_currency   sys_refcursor;
BEGIN
   OPEN v_currency
    FOR
       SELECT pub.title "Title",
              TO_CHAR (pub.price,
                       locale.currency_format,
                       'NLS_CURRENCY=' || locale.currency_symbol
                      ) "Price"
         FROM publication pub, locale
        WHERE pub.locale_id = locale.locale_id;

   RETURN v_currency;
END currency_conv_func;
```

I execute the currency conversion function as follows:

```
VARIABLE v_currency REFCURSOR
CALL currency_conv_func() INTO :v_currency;
PRINT v_currency
```

This returns the following list of prices:

```
Title                                                      Price
----------------------------------                         ----------------
Oracle PL/SQL Programming, 3rd Edition                     $54.95
Oracle PL/SQL プログラミング 基礎編 第 3 版                    ¥5,800
Oracle PL/SQL Programmierung, 2. Auflage                   €64
```

Note that no actual conversion is done here. If you need to automate the conversion from one currency to another, you will need to build your own rate table and conversion rules.

The NLS_ISO_CURRENCY symbol is generally a three-character abbreviation. With a few exceptions, the first two characters refer to the country or locale, and the third character represents the currency. For example, the United States dollar and the Japanese yen are USD and JPY, respectively. Many European countries use the euro, however, so the country/currency rule of thumb noted earlier cannot apply. It is simply represented as EUR.

The g11n schema includes ISO currency values to help us convert the prices of publications to their correct ISO abbreviations, as you can see in the ISO_CURRENCY_FUNC function:

```
/* File on web: g11n.sql */
FUNCTION iso_currency_func
   RETURN sys_refcursor
IS
   v_currency   sys_refcursor;
BEGIN
   OPEN v_currency
    FOR
       SELECT   title "Title",
                TO_CHAR (pub.price,
                         locale.iso_currency_format,
                         'NLS_ISO_CURRENCY=' || locale.iso_currency_name
                        ) "Price - ISO Format"
           FROM publication pub, locale
          WHERE pub.locale_id = locale.locale_id
       ORDER BY publication_id;

   RETURN v_currency;
END iso_currency_func;
```

To execute the ISO_CURRENCY_FUNC function, I run the following:

```
VARIABLE v_currency REFCURSOR
CALL iso_currency_func() INTO :v_currency;
PRINT v_currency
```

The result set shows the following:

```
Title                                                  Price - ISO Format
-------------------------------------                  ------------------
Oracle PL/SQL Programming, 3rd Edition                 USD54.95
Oracle PL/SQL プログラミング 基礎編 第３版                      JPY5,800
Oracle PL/SQL Programmierung, 2. Auflage               EUR64
```

USD, JPY, and EUR are included in my price display just as I expected based on the format mask.

Globalization Development Kit for PL/SQL

Starting with Oracle Database 10*g*, Oracle provides a Globalization Development Kit (GDK) for Java and PL/SQL that simplifies the g11n development process. If you are developing a multilingual application, determining the locale of each user and presenting locale-specific feedback may be the most difficult programming task you will face. The PL/SQL components of the GDK help with this aspect of g11n development, and are delivered in two packages: UTL_I18N and UTL_LMS.

UTL_118N Utility Package

The UTL_I18N package is the workhorse of the GDK. Its subprograms are summarized in Table 25-7.

Table 25-7. Programs in the UTL_I18N package

Name	Description
ESCAPE_REFERENCE	Allows you to specify an escape sequence for characters in the database that cannot be converted to the character set used in an HTMl or XML document. This function takes as input the source string and the character set of the HTML or XML document.
GET_COMMON_TIME_ZONES	Returns a list of the most common time zones. This is particularly useful when you present a user with a list of time zones he can select from to configure user settings.
GET_DEFAULT_CHARSET	Returns the default character set name or the email-safe name based on the language supplied to this function.
GET_DEFAULT_ISO_CURRENCY	Supplied with a territory, this function returns the appropriate currency code.
GET_DEFAULT_LINGUISTIC_SORT	Returns the most common sort for the supplied language.
GET_LOCAL_LANGUAGES	Returns local languages for a given territory.
GET_LOCAL_LINGUISTIC_SORTS	Returns a list of sort names based on a supplied language.
GET_LOCAL_TERRITORIES	Lists territory names based on a given language.
GET_LOCAL_TIMEZONES	Returns all time zones in a given territory.
GET_TRANSLATION	Translates the language and/or territory name into the specified language and returns the results.
MAP_CHARSET	Provides mapping between database character sets and email-safe character sets; particularly useful for applications that send data extracted from the database via email.
MAP_FROM_SHORT_LANGUAGE	Pass a short language name to this function, and it maps it to the Oracle language name.
MAP_LANGUAGE_FROM_ISO	Pass an ISO locale name to this function, and it returns the Oracle language name.
MAP_LOCALE_TO_ISO	Supply the Oracle language and territory to this function, and it returns the ISO locale name.
MAP_TERRITORY_FROM_ISO	Pass an ISO locale to this function, and it returns the Oracle territory name.
MAP_TO_SHORT_LANGUAGE	Reverse of the MAP_FROM_SHORT_LANGUAGE function; supply the full Oracle language name to this function, and it returns the short name.
RAW_TO_CHAR	Overloaded function that takes a RAW type as input and returns a VARCHAR2.
RAW_TO_NCHAR	Identical to RAW_TO_CHAR except the return value is of type NVARCHAR2.

Name	Description
STRING_TO_RAW	Converts either a VARCHAR2 or an NVARCHAR2 to the specified character set and returns a value of type RAW.
TRANSLITERATE	Performs script translation, based on transliteration name, for Japanese Kana.
UNESCAPE_REFERENCE	Reverse of the ESCAPE_REFERENCE function; recognizes escape characters and reverts them to their original characters.

The GET_LOCAL_LANGUAGES function is one of the most useful in this package. If I know a user's territory, I can reduce the list of values of valid languages for the user to choose from in an application using UTL_I18N.GET_LOCAL_LANGUAGES. This is great for applications in which the administrator must configure user-specific application settings. I can test it out using the following seed data:

```
CREATE TABLE user_admin (
   id NUMBER(10) PRIMARY KEY,
   first_name VARCHAR2(10 CHAR),
   last_name VARCHAR2(20 CHAR),
   territory VARCHAR2(30 CHAR),
   language VARCHAR2(30 CHAR))
/

BEGIN
INSERT INTO user_admin
     VALUES (1, 'Stan', 'Smith', 'AMERICA', 'AMERICAN');
INSERT INTO user_admin
     VALUES (2, 'Robert', 'Hammon', NULL, 'SPANISH');
INSERT INTO user_admin
     VALUES (3, 'Anil', 'Venkat', 'INDIA', NULL);
COMMIT;
END:
/
```

The territory is entered into the USER_ADMIN table. I can present a list of local languages for user Anil using the following anonymous block:

```
DECLARE
   -- Create array for the territory result set
   v_array   utl_i18n.string_array;
   -- Create the variable to hold the user record
   v_user    user_admin%ROWTYPE;
BEGIN
   -- Populate the variable with the record for Anil
   SELECT *
     INTO v_user
     FROM user_admin
    WHERE ID = 3;

   -- Retrieve a list of languages valid for the territory
   v_array := utl_i18n.get_local_languages (v_user.territory);
   DBMS_OUTPUT.put (CHR (10));
```

```
      DBMS_OUTPUT.put_line ('======================');
      DBMS_OUTPUT.put_line ('User: ' || v_user.first_name || ' '
                            || v_user.last_name
                           );
      DBMS_OUTPUT.put_line ('Territory: ' || v_user.territory);
      DBMS_OUTPUT.put_line ('======================');

      -- Loop through the array
      FOR y IN v_array.FIRST .. v_array.LAST
      LOOP
         DBMS_OUTPUT.put_line (v_array (y));
      END LOOP;
   END;
```

This returns the following:

```
   ======================
   User: Anil Venkat
   Territory: INDIA
   ======================
   ASSAMESE
   BANGLA
   GUJARATI
   HINDI
   KANNADA
   MALAYALAM
   MARATHI
   ORIYA
   PUNJABI
   TAMIL
   TELUGU
```

This list of values is much easier for a user to manage than a complete list of all languages. The same can be done for territories where the language is known. Suppose that Robert currently has a NULL territory, but his language is specified as SPANISH. The following anonymous block returns a list of valid territories for the SPANISH language:

```
   DECLARE
      -- Create array for the territory result set
      v_array   utl_i18n.string_array;
      -- Create the variable to hold the user record
      v_user    user_admin%ROWTYPE;
   BEGIN
      -- Populate the variable with the record for Robert
      SELECT *
        INTO v_user
        FROM user_admin
       WHERE ID = 2;

      -- Retrieve a list of territories valid for the language
      v_array := utl_i18n.get_local_territories (v_user.LANGUAGE);
      DBMS_OUTPUT.put (CHR (10));
      DBMS_OUTPUT.put_line ('======================');
      DBMS_OUTPUT.put_line ('User: ' || v_user.first_name || ' '
                             || v_user.last_name
                            );
      DBMS_OUTPUT.put_line ('Language: ' || v_user.LANGUAGE);
      DBMS_OUTPUT.put_line ('=======================');

      -- Loop through the array
      FOR y IN v_array.FIRST .. v_array.LAST
      LOOP
         DBMS_OUTPUT.put_line (v_array (y));
      END LOOP;
   END;
```

The output is:

```
=======================
User: Robert Hammon
Language: SPANISH
=======================
SPAIN
CHILE
COLOMBIA
COSTA RICA
EL SALVADOR
GUATEMALA
MEXICO
NICARAGUA
PANAMA
PERU
PUERTO RICO
VENEZUELA
```

With a territory, I can present a list of valid languages, time zones, and currencies to the end user and make configuration much easier. Once a language is selected, I can get the default character set, the default linguistic sort, the local territories, and the short language name, all using UTL_I18N.

UTL_LMS Error-Handling Package

UTL_LMS is the second package that is part of the GDK. It includes two functions that retrieve and format an error message:

GET_MESSAGE
: Returns the raw message based on the language specified. By raw message, I mean that any parameters required for the message are not included in what is returned by GET_MESSAGE.

FORMAT_MESSAGE
: Adds detail to the message.

Consider the following example:

```
DECLARE
   v_bad_bad_variable   PLS_INTEGER;
   v_function_out       PLS_INTEGER;
   v_message            VARCHAR2 (500);
BEGIN
   v_bad_bad_variable := 'x';
EXCEPTION
   WHEN OTHERS
   THEN
      v_function_out :=
               utl_lms.GET_MESSAGE (06502, 'rdbms', 'ora', NULL, v_message);
      -- Output unformatted and formatted messages
      DBMS_OUTPUT.put (CHR (10));
      DBMS_OUTPUT.put_line ('Message - Not Formatted');
      DBMS_OUTPUT.put_line ('=======================');
      DBMS_OUTPUT.put_line (v_message);
      DBMS_OUTPUT.put (CHR (10));
      DBMS_OUTPUT.put_line ('Message - Formatted');
      DBMS_OUTPUT.put_line ('===================');
      DBMS_OUTPUT.put_line (utl_lms.format_message (v_message,
                                                     ': The quick brown fox'
                                                    )
                           );
END;
```

In the call to UTL_LMS.GET_MESSAGE, the value for language was left to the default, NULL. In this case, the returned message will be in the default language, determined by NLS_LANGUAGE. My instance returns:

```
Message - Not Formatted
=======================
PL/SQL: numeric or value error%s

Message - Formatted
===================
PL/SQL: numeric or value error: The quick brown fox
```

Because a language value can be passed to UTL_LMS.GET_MESSAGE, I simply pass the application user's language when getting the message.

GDK Implementation Options

The GDK functions allow for several different implementation options. If you are supporting only two or three locales, it might be easiest to separate your implementation by locale. For your German system, set your database and application servers to the appropriate settings for that locale, and have a completely separate environment for your users in France. More often than not, however, you will want to look at a true multilingual environment in which you can add new locales without purchasing and configuring a separate environment. This requires extra development effort up front, but is much easier to manage in the long run.

The method by which you determine a user's locale depends largely on who your users are and the type of application you are developing. In the following subsections I discuss three options for you to consider in your design.

Method 1: Locale buttons

As a frequent surfer (of the Internet, of course), I regularly see this method in action on web pages. Visit a company that does business in different locales, and you might see buttons or links on the main page that look like the following:

> in English | en Español | en Français | in Italiano

This is great for web pages in which the business is restricted to a few locations and where date/time accuracy is not required. Users can choose their language and currency settings through the simple act of clicking the appropriate link, and should they choose incorrectly, they can simply use the back button on the browser to correct the problem.

In this scenario, you can either have separate pages and code for each locale or store session-specific selections in cookies or the database so all localization is controlled by the database. The most common approach here is to allow the application tier to control localization.

Method 2: User administration

Applications that have a managed user base (not open to anonymous Internet users, for example) will find Method 2 a great way to control locale settings. For a small number of users, it is possible to use the UTL_I18N package to deliver a list of values (LOV) showing available time zones, locales, languages, and territories, as I demonstrated earlier. The user or administrator simply selects the settings that are appropriate for the user, and every time that user logs in, that application reads these settings and delivers the appropriate localizations.

What about instances where there are a lot of users? It isn't feasible to manage each user's settings individually in all cases. In this case, we can take a lesson from the designers of the Oracle database (which is a global application, right?) and create profiles. Add to your application the ability to create a profile, and assign locale settings to it rather than to users. When you add a user, simply assign the profile. This cuts out many of the administrative headaches, especially if you ever have to go back and make a change later. You can simply change the profile rather than having to change all users.

Method 3: Hybrid

Method 3 is a combination of Methods 1 and 2. It is used frequently with Internet applications that have online stores. Most customers begin by browsing a site to see if it has something that they want. At this point, requiring them to fill out details about their location is premature, but they do need locale-specific features such as data sorted and displayed in their language and in the correct currency format. To make certain that basic locale information is correct, offer the solution discussed in Method 1.

Once the decision to purchase is made, however, it is quite appropriate to have the user enter a user profile including locale-specific information. The localization becomes more precise, using exact date/time and currency information from the database, all based on the customer's locale.