Oracle® Application Server

Globalization Guide

10*g* Release 3 (10.1.3.1.0)

B31263-01

July 2006

ORACLE®

Oracle Application Server Globalization Guide, 10*g* Release 3 (10.1.3.1.0)

B31263-01

Primary Author: Caroline Johnston

Contents

4 Implementing HTML Features

5 Using a Centralized Database

6 Configuring Oracle Application Server for Global Deployment

7 A Multilingual Demonstration for Oracle Application Server

List of Examples

List of Tables

Preface

Oracle Application Server Globalization Guide describes how to design, develop, and deploy Internet applications for a global audience.

This preface contains the following topics:

Audience

Oracle Application Server Globalization Guide is intended for Internet application developers and Webmasters who design, develop, and deploy Internet applications for a global audience.

To use this document, you need to have some programming experience and be familiar with Oracle databases.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documentation

For more information, see these Oracle resources:

- The Oracle Application Server documentation set
- *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library

Printed documentation is available for sale in the Oracle Store at

`http://oraclestore.oracle.com/`

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

`http://www.oracle.com/technology/membership/`

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

`http://www.oracle.com/technology/documentation/`

Conventions

This section describes the conventions used in the text and code examples of this documentation set. The following table describes the conventions in this document to help you more quickly identify special terms:

Convention	Meaning	Example
. . .	Ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS` *`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT` *`col1`*`,` *`col2`*`, ... ,` *`coln`* `FROM employees;`
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`* `[ ,` *`precision`* `])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`
|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}` `[COMPRESS | NOCOMPRESS]`

Convention	Meaning	Example
Bold	Bold typeface indicates: ■ Terms that are defined in the text ■ Terms that appear in a glossary A graphical user interface (GUI) element that has an action associated with it.	When you specify this clause, you create an **index-organized table**. Click **OK** to continue.
Italics	Italic typeface indicates book titles or emphasis.	*Oracle Database Concepts* Ensure that the recovery catalog and target database do *not* reside on the same disk.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements. Such elements include computer and database names, net service names, and connect identifiers, as well as user-supplied database objects and structures, column names, packages and classes, usernames and roles, program units, and parameter values. **Note:** Some programmatic elements use a mixture of UPPERCASE and lowercase. Enter these elements as shown.	Enter `sqlplus` to open SQL*Plus. The password is specified in the `orapwd` file. Back up the data files and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table. Set the `QUERY_REWRITE_ENABLED` initialization parameter to `true`. Connect as `oe` user. The `JRepUtil` class implements these methods.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.
Other notation	You must enter symbols other than brackets, braces, vertical bars, and ellipsis points as shown.	`acctbal NUMBER(11,2);` `acct CONSTANT NUMBER(4) := 3;`
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system. Such elements include parameters, privileges, datatypes, Oracle Recovery Manager keywords, SQL keywords, SQL*Plus or utility commands, packages and methods, as well as system-supplied column names, database objects and structures, usernames, and roles. However, these terms are not case sensitive, so you can enter them in lowercase.	You can specify this clause only for a `NUMBER` column. You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.

1

Overview of Globalization in Oracle Application Server

This chapter contains the following topics:

Internet Applications Globalization

It is important for businesses to make their Internet applications available to users around the world with appropriate locale characteristics, such as language and currency formats. Oracle Application Server is fully internationalized to provide a global platform for developing and deploying Internet applications.

Building an Internet application or Web site for Oracle Application Server requires good globalization practices in development and deployment. This book describes recommended globalization practices.

Globalization Concepts

You need to be familiar with the following concepts to understand globalization:

Locale

Locale refers to a language, a character set, and the region (territory) in which the language is spoken. Information about the region includes formats for dates and currency. For example, the primary languages of the United States and Great Britain are both forms of English, but the two territories have different currencies and different conventions for date formats. Therefore, the United States and Great Britain are different locales.

Character Set

A **character set** defines the binary values associated with the characters that make up a language. For example, the ISO-8859-1 character set can be used to encode most Western European languages.

Unicode

Unicode is a universal character set that defines binary values for characters in almost all languages. Unicode characters can be encoded as follows:

- In 1 to 4 bytes in the UTF-8 character set
- In 2 or 4 bytes in the UTF-16 character set
- In 4 bytes in the UTF-32 character set

Designing a Global Internet Application

There are several approaches to designing global Internet applications. This book discusses the following approaches:

- Monolingual

  You can design a monolingual Internet application so it supports several instances. Each instance supports a different locale. Users with different locale preferences must invoke the instance that serves their locale.

- Multilingual

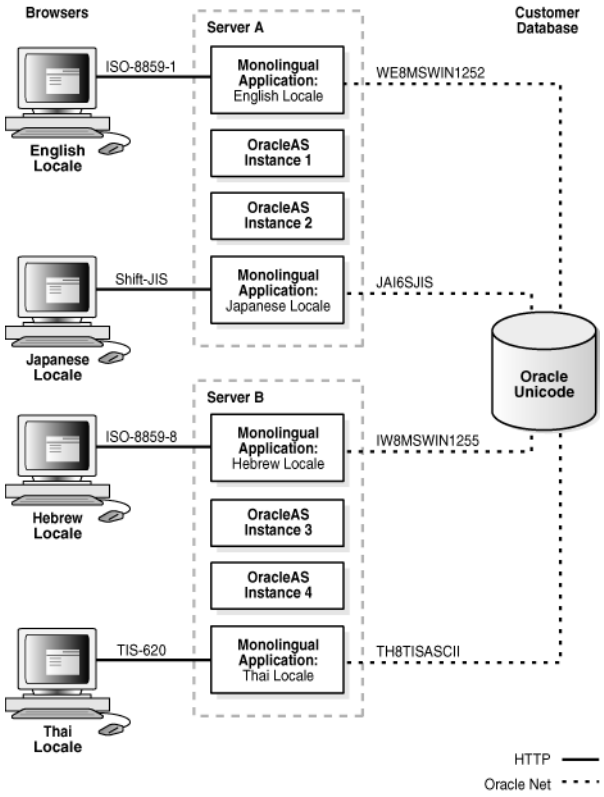
  You can design a multilingual Internet application to support several locales with one instance. All users, regardless of locale, can invoke the same instance.

Both designs include one centralized database that uses a Unicode character set.

Monolingual Internet Application Architecture

Figure 1–1 shows the design of a monolingual Internet application.

Figure 1–1 Monolingual Internet Application Architecture



The clients (in English, Japanese, Hebrew, and Thai locales) communicate with separate instances of Oracle Application Server through HTTP connections. One instance of the application runs in the same locale as one of the Oracle Application Server instances. For example, the English application runs in the same locale as Oracle Application Server Instance 1. The English and Japanese applications and their Oracle Application Server instances are running on Server A, and the Hebrew and Thai applications and their instances are running on Server B. Each Oracle Application Server instance communicates with the Unicode database. The instances communicate with the database through Oracle Net.

The client character set for the English locale, for example, is ISO-8859-1. The Oracle Application Server instance that is associated with the English locale, Instance 1, uses the Oracle character set WE8MSWIN1252 to communicate with the database. The database character set is a Unicode character set.

See Also: Chapter 5, "Using a Centralized Database"

Table 1–1 shows the advantages and disadvantages of deploying monolingual Internet applications. As the number of locales increases, the disadvantages outweigh the advantages of the monolingual design. This type of application design is suitable for customers who support only one or two locales.

Table 1–1 Advantages and Disadvantages of Monolingual Internet Application Design

Advantages	Disadvantages
You can separate the support of different locales into different servers. This allows locales to be supported in different time zones. Work load can be distributed accordingly.	There are more Oracle Application Server servers to administer.

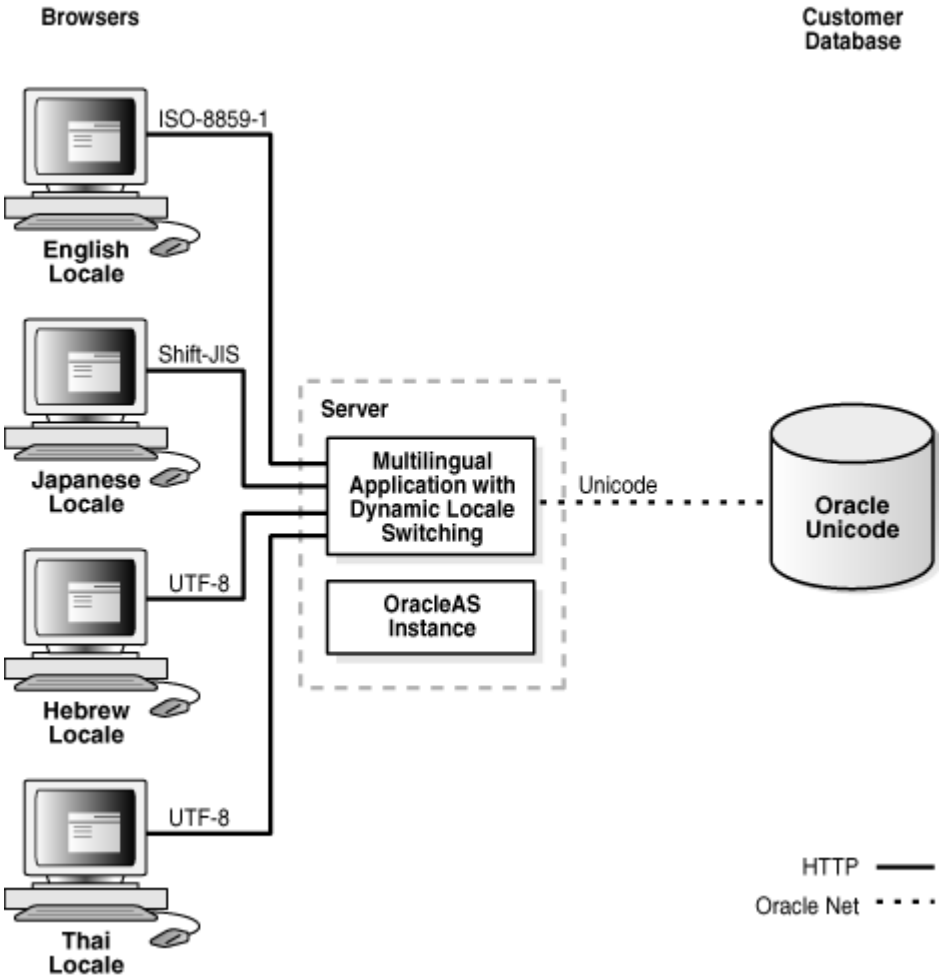
Table 1–1 (Cont.) Advantages and Disadvantages of Monolingual Internet Application

Advantages	Disadvantages
Writing the code is simpler than for a multilingual Internet application.	The Internet application requires more testing resources to certify it on each Oracle Application Server instance. You must configure Oracle Application Server for each instance of the application. You must maintain a server instance for each locale regardless of the amount of work that is demanded of it. Load-balancing is possible only among a group of Oracle Application Server instances that support the same locale. Supporting multilingual content is difficult.

Multilingual Internet Application Architecture

Figure 1–2 shows the design of a multilingual Internet application.

Figure 1–2 Multilingual Internet Application Architecture



The clients (in English, Japanese, Hebrew, and Thai locales) communicate with one Oracle Application Server instance through HTTP connections. Each client can use a

different character set because each application running on Oracle Application Server is configured to support several locales simultaneously, regardless of the locale of the Oracle Application Server instance. The Oracle Application Server instance and the database communicate through Oracle Net. Both the application running on the Oracle Application Server instance and the database use Unicode character sets.

See Also: Chapter 5, "Using a Centralized Database"

In order to support several locales in a single application instance, an application should:

- Process character data in Unicode so that it can support data in any language
- Dynamically detect the user's locale and adapt to the locale by constructing HTML pages in the correct language and cultural conventions
- Dynamically determine the character set to use for HTML pages and convert content to and from Unicode to the HTML page encoding

Table 1–2 shows the advantages and disadvantages of deploying multilingual Internet applications.

Table 1–2 Advantages and Disadvantages of Multilingual Internet Application Design

Advantages	Disadvantages
You can use one Oracle Application Server configuration, which reduces maintenance costs. Performance tuning and capacity planning do not depend on the number of locales. Supporting additional languages is relatively easy. You do not need to add more machines for the new locales. You can test the application for several locales in a single testing environment. The application can support multilingual content.	Multilingual applications are more complex to code than monolingual applications. They must be able to detect locales dynamically and use Unicode. This is costly if you only need to support one or two languages.

Overview of Developing Global Internet Applications

Building an Internet application for Oracle Application Server that supports different locales requires good development practices. The application itself must be aware of the user's locale and be able to present locale-appropriate content to the user. Clients must be able to communicate with the application server regardless of the client's locale, with minimal character set conversion. The application server must be able to access the database server with data in many languages, again with minimal character set conversion. Character set conversion decreases performance and increases the chance of data loss because some characters may not be available in the target character set.

See Also: *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library for more information about character set conversion

Note: In this book, **encoding** and **page encoding** refer to the character set used in a particular programming environment.

Oracle Application Server also supports the development of global applications using the following Oracle Application Server components:

- Oracle Application Server Forms Services
- Oracle Application Server Reports Services
- Oracle Business Intelligence Discoverer
- Oracle Web Cache
- Oracle Application Server Wireless
- Oracle Globalization Development Kit

This guide discusses global application development in terms of language and development environments. It addresses the basic tasks associated with developing and deploying global Internet applications, including developing locale awareness, implementing HTML features, accessing a centralized database, and configuring Oracle Application Server.

2

Developing Locale Awareness

This chapter contains the following topics:

Developing Locale Awareness in Global Internet Applications

Global Internet applications need to be aware of the user's locale.

Locale-sensitive functions, such as date formatting, are built into programming environments such as C/C++, Java, and PL/SQL. Applications can use locale-sensitive functions to format the HTML pages according to the cultural conventions of the user's locale.

Different programming environments represent locales in different ways. For example, the French (Canada) locale is represented as follows:

Environment	Representation	Locale	Explanation
Various	ISO standard	`fr-CA`	`fr` is the language code defined in the ISO 639 standard. `CA` is the country code defined in the ISO 3166 standard.
Java	Java locale object	`fr_CA`	Java uses the ISO language and country code.
C/C++	POSIX locale name	`fr_CA` on Sun Solaris	POSIX locale names may include a character set that overrides the default character set. For example, the de.ISO8859-15 locale is used to support the Euro symbol.
PL/SQL and SQL	`NLS_LANGUAGE` and `NLS_TERRITORY` parameters	`NLS_LANGUAGE= "CANADIAN FRENCH"` `NLS_TERRITORY= "CANADA"`	**See Also:** "Configuring the NLS_LANG Parameter" in Chapter 6

Table 2–1 shows how different programming environments represent some commonly used locales.

Table 2–1 Locale Representations in Different Programming Environments

Locale	ISO	Java	POSIX Solaris	NLS_LANGUAGE, NLS_TERRITORY
Arabic (U.A.E.)	ar	ar	ar	ARABIC, UNITED ARAB EMIRATES
Chinese (P.R.C)	zh-CN	zh_CN	zh_CN	SIMPLIFIED CHINESE, CHINA
Chinese (Taiwan)	zh-TW	zh_TW	zh_TW	TRADITIONAL CHINESE, TAIWAN
English (U.S.A)	en	en_US	en_US	AMERICAN, AMERICA
English (United Kingdom)	en-GB	en_GB	en_UK	ENGLISH, UNITED KINGDOM
French (Canada)	fr-CA	fr_CA	fr_CA	CANADIAN FRENCH, CANADA
French (France)	fr	fr_FR	fr	FRENCH, FRANCE
German (Germany)	de-DE	de_DE	de	GERMAN, GERMANY
Greek	el	el	el	GREEK, GREECE
Hebrew	he	he	he	HEBREW, ISRAEL
Italian (Italy)	it	it	it	ITALIAN, ITALY
Japanese	ja-JP	ja_JP	ja_JP	JAPANESE, JAPAN
Korean	ko-KR	ko_KR	ko_KR	KOREAN, KOREA
Portuguese (Brazil)	pt-BR	pt_BR	pt_BR	BRAZILIAN PORTUGUESE, BRAZIL
Portuguese (Portugal)	pt	pt	pt	PORTUGUESE, PORTUGAL
Spanish (Spain)	es-ES	es_ES	es	SPANISH, SPAIN
Thai	th	th	th	THAI, THAILAND
Turkish	tr	tr	tr	TURKISH, TURKEY

If you write applications for more than one programming environment, then locales must be synchronized between environments. For example, Java applications that call PL/SQL procedures should map the Java locales to the corresponding `NLS_LANGUAGE` and `NLS_TERRITORY` values and change the parameter values to match the user's locale before calling the PL/SQL procedures.

There are two things that affect an application's overall locale awareness: the development environment in which you create the application, and the target architecture for which the application is built. This chapter addresses these topics with respect to both monolingual and multilingual application architectures.

Determining a User's Locale in Monolingual Internet Applications

A monolingual application, by definition, serves users with the same locale. A user's locale is fixed in a monolingual application and is the same as the default runtime locale of the programming environment.

In most programming environments, almost all locale-sensitive functions implicitly use the default runtime locale to perform their tasks. Monolingual applications can rely on this behavior when calling these functions.

Determining a User's Locale in Multilingual Internet Applications

In a multilingual application, the user's locale may vary. Multilingual applications should do the following:

- Dynamically detect the user's locale
- Construct HTML content in the language of the locale
- Use the cultural conventions implied by the locale

Multilingual applications can determine a user's locale dynamically in the following ways:

- Based on the user profile information from an LDAP directory server such as Oracle Internet Directory

  The application can store the user profile in the Oracle Internet Directory server provided by Oracle Application Server. The LDAP schema for the user profile should include a preferred locale attribute. This method does not work if a user has not logged on before.

- Based on the default ISO locale of the user's browser

  Every HTTP request sends the default ISO locale of the browser with the Accept-Language HTTP header. If the Accept-Language header is NULL, then the locale should default to English. The drawback of this approach is that the Accept-Language header may not be a reliable source of information about the user's locale.

- Based on user input

  Users can select a locale from a list or a group of icons such as flags.

You can use these methods of determining the user's locale together or separately. After the application determines the locale, the locale should be:

- Mapped to the locale representations that correspond to the programming environments on which the application runs
- Used in locale-sensitive functions

> **See Also:** Table 2–1 for common locale representations in different programming environments

Locale Awareness in J2EE and Internet Applications

This section discusses locale awareness in terms of the particular programming language and development environment in which an application is written.

Locale Awareness in Java Applications

A Java locale object represents the corresponding user's locale in Java. The Java encoding used for the locale is required to properly convert Java strings to and from byte data.

Consider the Java encoding for the locale when you make the Java code aware of a user's locale. There are two ways to make a Java method sensitive to the Java locale and the Java encoding:

- Using the default Java locale and default Java encoding for the method
- Explicitly specifying the Java locale and Java encoding for the method

Locale Awareness in Monolingual Java Applications

Monolingual applications should run implicitly with the default Java locale and default Java encoding so that the applications can be configured easily for a different locale. For example, to create a date format using the default Java locale, use the following method call:

```
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL);
dateString = df.format(date); /* Format a date */
```

Locale Awareness in Multilingual Java Applications

You should develop multilingual applications that are independent of fixed default locales or encodings. Explicitly specify the Java locale and Java encoding that correspond to the current user's locale. For example, specify the Java locale object that corresponds to the user's locale, identified by `user_locale`, in the `getDateTimeInstance()` method:

```
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL, user_
locale);
dateString = df.format(date); /* Format a date */
```

Note: The only difference between the example code for the monolingual application and the multilingual application is the inclusion of `user_locale`.

Similarly, do not use encoding-sensitive methods that assume the default Java encoding. For example, you should not use the `String.getBytes()` method in a multilingual application because it is encoding-sensitive. Instead, use the method that accepts encoding as an argument, which is `String.getBytes(String encoding)`. Be sure to specify the encoding used for the user's locale.

Do not use the `Locale.setDefault()` method to change the default locale for the following reasons:

- It changes the Java default locale for all threads and makes your applications unsafe to threads
- It does not affect the Java default encoding

The Oracle Globalization Development Kit for Java provides a set of Java classes to ensure consistency on locale-sensitive behaviors with Oracle databases. It provides an application framework that enables you to use the locale determination methods declaratively.

See Also: Chapter 3, "Oracle Globalization Development Kit"

Locale Awareness in Perl and C/C++ Applications

Perl and C/C++ use the POSIX locale model for internationalized applications.

Locale Awareness in Monolingual Perl and C/C++ Applications

Monolingual applications should be sensitive to the default POSIX locale, which is configured by changing the value of the LC_ALL environment variable or changing the operating system locale from the Control Panel in Microsoft Windows.

See Also: Table 2–1 for a list of commonly used POSIX locales

To run on the default POSIX locale, the applications should call the `setlocale()` function to set the default locale to the one defined by LC_ALL and use the POSIX locale-sensitive functions such as `strftime()` thereafter. Note that the `setlocale()` function affects the current process and all threads associated with it, so any multithread application should assume the same POSIX locale in each thread. The following example gets the current time in the format specific to the default locale in Perl:

```
use locale;
use POSIX qw (locale_h);
...
$old_locale = setlocale( LC_ALL, "" );
$dateString = POSIX::strftime( "%c", localtime());
...
```

Locale Awareness in Multilingual Perl and C/C++ Applications

Multilingual applications should be sensitive to dynamically determined locales. Call the `setlocale()` function to initialize the locale before calling locale-sensitive functions. For example, the following C code gets the local time in the format of the user locale identified by `user_locale`:

```
#include <locale.h>
#include <time.h>
    ...
    const char *user_locale = "fr";
    time_t ltime;
    struct tm *thetime;
    unsigned char dateString[100];
    ...
    setlocale(LC_ALL, user_locale);
    time (&ltime);
    thetime = gmtime(&ltime);
    strftime((char *)dateString, 100, "%c", (const struct tm *)thetime));
    ...
```

You must map user locales to POSIX locale names for applications to initialize the correct locale dynamically in C/C++ and Perl. The POSIX locales depend on the operating system.

Locale Awareness in SQL and PL/SQL Applications

PL/SQL procedures run in the context of a database session whose locale is initialized by the NLS_LANG parameter in the **database access descriptor (DAD)**. The NLS_LANG parameter specifies top-level globalization parameters, NLS_LANGUAGE and NLS_TERRITORY, for the database session. Other globalization parameters, such as NLS_SORT and NLS_DATE_LANGUAGE, inherit their values from these top-level parameters. These globalization parameters define the locale of a database session.

There are two ways to make SQL and PL/SQL functions locale sensitive:

- Basing the locale on the globalization parameters of the current database session
- Explicitly specifying the globalization parameters

See Also:

- "Configuring the NLS_LANG Parameter" in Chapter 6
- *Oracle Database Reference 10g Release 1 (10.1)* in the Oracle Database Documentation Library
- *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library

for more information about globalization parameters

Locale Awareness in Monolingual SQL and PL/SQL Applications

Generally speaking, the initial values of the globalization parameters inherited from NLS_LANG are sufficient for monolingual PL/SQL procedures. For example, the following PL/SQL code calls the TO_CHAR( ) function to get the formatted date, which uses the current values of the NLS_DATE_FORMAT and NLS_DATE_LANGUAGE parameters:

```
mydate date;
dateString varchar2(100);
...
select sysdate into mydate from dual;
dateString = TO_CHAR(mydate);
```

If the initial values of the globalization parameters are not appropriate, then use an ALTER SESSION statement to overwrite them for the current database session. You can use the ALTER SESSION statement with the DBMS_SQL package. For example:

```
cur integer;
status integer;
...
cur := dbms_sql.open_cursor;
dbms_sql.parse(cur, 'alter session set nls_date_format = "Day Month, YYYY"',
       dbms_sql.native);
status := dbms_sql.execute(cur);
```

Locale Awareness in Multilingual SQL and PL/SQL Applications

Multilingual applications should use ALTER SESSION statements to change the locale of the database session to the user's locale before calling any locale-sensitive SQL or PL/SQL functions. You can use the ALTER SESSION statement with the DBMS_SQL package. For example:

```
cur integer;
status integer;
...
cur := dbms_sql.open_cursor;
dbms_sql.parse(cur, 'alter session set nls_language = "NLS_LANGUAGE_of_user_
       locale"', dbms_sql.native);
dbms_sql.parse(cur, 'alter session set nls_territory = "NLS_TERRITORY_of_
       user_locale"', dbms_sql.native);
status := dbms_sql.execute(cur);
```

Alternatively, applications can specify the globalization parameters in every SQL function that accepts a globalization parameter as an argument. For example, the following PL/SQL code gets a date string based on the language of the user's locale:

```
mydate date;
dateString varchar2(100);
...
select sysdate into mydate from dual;
dateString TO_CHAR(mydate, 'DD-MON-YYYY HH24:MI:SSxFF',
                    'NLS_DATE_LANGUAGE=language' );
...
```

In the preceding code example, *language* specifies the Oracle language name for the user's locale.

Locale Awareness in Oracle Application Server Component Applications

This section discusses locale awareness in terms of application development for particular Oracle Application Server components.

Locale Awareness in Oracle Application Server Wireless Services

Oracle Application Server Wireless (OracleAS Wireless) sends all mobile context information as HTTP headers when invoking a request. The user locale is sent using the `X-Oracle-User.Locale` header. The locale value contains the ISO language, and an optional ISO country code, separated by a hyphen. For example, "en-US", "zh-CN", and "ja" are all valid locale values for this header. Mobile service applications should use the user locale specified in this header to determine the language and cultural conventions used in the user interface.

For example, JSP applications may retrieve the user locale as follows:

```
<%
   String userLocale = request.getHeader("X-Oracle-User.Locale");
%>
```

Locale Awareness in Oracle Application Server Forms Services

The Oracle Application Server Forms Services (OracleAS Forms Services) architecture includes:

- A Java client (browser)
- OracleAS Forms Services (middle tier)
- The Oracle customer database (back end)

The Java client is dynamically downloaded from Oracle Application Server when a user runs an OracleAS Forms Services session. The Java client provides the user interface for the OracleAS Forms Services Runtime Engine. It also handles user interaction and visual feedback for actions such as navigating between items or checking a checkbox.

OracleAS Forms Services consists of the OracleAS Forms Services Runtime Engine and the Forms Listener Servlet. The OracleAS Forms Services Runtime Engine is the process that maintains a connection to the database on behalf of the Java client. The Forms Listener Servlet acts as a broker, taking connection requests from the Java client processes and initiating a OracleAS Forms Services runtime process on their behalf.

The `NLS_LANG` parameter for OracleAS Forms Services initializes the locale of OracleAS Forms Services. The `NLS_LANGUAGE` parameter derives its value from `NLS_LANG` and determines the language of OracleAS Forms Services messages. The `NLS_TERRITORY` parameter also derives its value from `NLS_LANG` and determines conventions such as date and currency formats.

By default, the NLS_LANG parameter for OracleAS Forms Services initializes the Java client locale. The locale of the Java client determines such things as button labels on default messages and parts of strings in menus.

> **See Also:** *Oracle Application Server Forms Services Deployment Guide*

Locale Awareness in Monolingual OracleAS Forms Services Applications

A user's locale is fixed in a monolingual OracleAS Forms Services application and is usually the same as the default OracleAS Forms Services locale. When you develop a monolingual OracleAS Forms Services application, you must develop it to conform to the intended user's locale. The database character set should be a superset of the OracleAS Forms Services character set.

For example, a monolingual Forms Services application for a Japanese locale should include Japanese text, Japanese button labels, and Japanese menus. The application should also connect to a database whose character set is JA16SJIS, JA16EUC, or UTF8.

The NLS_LANG parameter in the `default.env` file controls the Forms Services locale. Additionally, in order to pass non-Latin-1 parameters to Forms Services, you can set the `defaultcharset` parameter in `formsweb.cfg`.

> **See Also:** *Oracle Application Server Forms Services Deployment Guide*

Locale Awareness in Multilingual OracleAS Forms Services Applications

In a multilingual environment, the application can dynamically determine the locale of OracleAS Forms Services in two ways:

- Based on the user's profile
- Based on the user's input

When you develop an OracleAS Forms Services application you must choose one of these methods.

You can configure multilingual OracleAS Forms Services applications by using multiple environment configuration files (`envFile`). For example, do the following to create a form and translate it into different languages:

1. Create a form called `form.fmx`.
2. Translate it into Japanese and into Arabic using Oracle Translator.
3. Save them as `d:\form\ja\form.fmx` (Japanese) and `d:\form\ar\form.fmx` (Arabic).
4. Create two environment configurations files, `ja.env` and `ar.env`.
5. Specify the following settings in the appropriate environment file as follows:

Form	Environment File	NLS_LANG	FORMS_PATH
d:\form\ja\form.fmx	ja.env	JAPANESE_JAPAN.JA16SJIS	d:\form\ja
d:\form\ar\form.fmx	ar.env	ARABIC_EGYPT.ARMSWIN1256	d:\form\ar

Also, you can configure OracleAS Forms Services to read the preferred language settings of the browser. For example, if you have a human resources (HR) application translated into 24 languages, then add an application entry in the `formsweb.cfg` file similar to the following:

```
[HR]
default.env
[HR.de]
de.env
[HR.fr]
fr.env
[HR.it]
it.env
.
.
.
```

When the Forms Servlet detects a language preference in the browser, it checks the `formsweb.cfg` file to see if there is a translated version of the application.

For example, suppose the request is `http://myserver.mydomain/forms/frmservlet?config=HR` and the preferred languages are set to German (`de`), Italian (`it`), and French (`fr`), in this order, then this is the order of priority. The Forms Servlet tries to read from the application definitions in the following order:

```
HR.de
HR.it
HR.fr
HR
```

If the Forms Servlet cannot find any of those configurations, then it uses the `HR` configuration (`default.env`).

This means that you can configure Oracle Forms to support multiple languages with one URL. Each application definition can have its own environment file that contains the NLS language parameter definition. You can also specify separate working directory information and path information for each application.

> **See Also:** Chapter 6, "Configuring Oracle Application Server for Global Deployment"

Additionally, Oracle Forms can display more than one language in a form if the Java client machine has the Albany WT J font installed. You can obtain this font from the utilities CD-ROM in your CD pack or from

`http://metalink.oracle.com`

The Albany WT J font should be copied to `%WINDOWS%\Fonts` directory if the client is using JInitiator 1.3.1, or `%JAVA_HOME%\lib\fonts` directory if the client is using Java Plug-in 1.4.1.

Locale Awareness in Oracle Reports

The Oracle Reports (OracleAS Reports Services) architecture includes:

- A client tier (browser)
- A Reports Server (middle tier)
- An Oracle customer database (back end)

OracleAS Reports Services can run multiple reports simultaneously upon users' requests. The Reports Server enters requests for reports into a job queue and dispatches them to a dynamic, configurable number of pre-spawned runtime engines.

The runtime engine connects to the database, retrieves data, and formats output for the client.

The NLS_LANG setting for the Reports Server initializes the locale of the runtime engine. The NLS_LANGUAGE parameter derives its value from the NLS_LANG parameter and determines the language of the Reports Server messages. The NLS_TERRITORY parameter derives its value from the NLS_LANG parameter and determines the date and currency formats. For example, if NLS_LANG is set to JAPANESE_JAPAN.JA16SJIS, then Reports Server messages are in Japanese and reports use the Japanese date format and currency symbol.

The dynamic environment switching feature enables one instance of the Reports Server to serve reports with any arbitrary environment setting, including NLS_LANG. Using the environment element in the Reports Server configuration file, you can create a language environment that can be referenced in two ways:

- On a per-runtime engine basis through the engine element of the Reports Server configuration file
- On a per-job basis using the ENVID command line argument

> **See Also:** *Oracle Application Server Reports Services Publishing Reports to the Web* for more information about dynamic environment switching

Report output is generated in the OracleAS Reports Services character set. The client needs to be aware of the character set in which OracleAS Reports Services generated the HTML or XML.

> **See Also:** "Specifying the Page Encoding in Oracle Application Server Reports Services Applications" in Chapter 4

Locale Awareness in Monolingual OracleAS Reports Services Applications

A user's locale is fixed in a monolingual OracleAS Reports Services application and is usually the same as the locale of the Reports Server. The database character set should be a superset of the Reports Server character set.

Locale Awareness in a Multilingual OracleAS Reports Services Application

In a multilingual report, the application can dynamically determine the locale of the Reports Server in two ways:

- Based on the user's profile
- Based on the user's input

When you develop a report you must choose one of these methods.

You can use the dynamic environment switching feature to support multiple languages.

> **See Also:**
>
> - "Specifying the Page Encoding in HTML for Oracle Application Server Reports Services" in Chapter 4 for more information about the encoding of HTML, XML, and JSP report output
> - "Configuring OracleAS Reports Services for Global Deployment" in Chapter 6 for more information about specifying NLS_LANG parameters from the command line
> - *Oracle Application Server Reports Services Publishing Reports to the Web* for more information about dynamic environment switching

Locale Awareness in Oracle Business Intelligence Discoverer

Oracle Business Intelligence Discoverer (OracleBI Discoverer) can simultaneously support users with different locales. Users may explicitly control the locale used for the user interface, or they may allow OracleBI Discoverer to automatically determine the default. The order of precedence for determining the language and locale is:

1. Language and locale settings included in the URL for OracleBI Discoverer.
2. Language and locale settings specified in the OracleBI Discoverer Connection. If the locale is specified in the user's browser, then the language settings in the user's browser is used.
3. Language and locale of Oracle Application Server.

For example, suppose a user's browser's language and locale are set to `German - Germany` and the user goes to the URL to start OracleBI Discoverer. The HTML page returned to the user is displayed in German. If the user clicks on the OracleBI Discoverer Connection, which has the language and locale specified as `English - US`, the OracleBI Discoverer user interface appears in English. This is because the OracleBI Discoverer Connection settings take precedence over the browser's settings.

3

Oracle Globalization Development Kit

This chapter includes the following sections:

Overview of the Oracle Globalization Development Kit

The Oracle Globalization Development Kit (GDK) simplifies the development process and reduces the cost of developing Internet applications used to support a global environment.

The GDK includes comprehensive programming APIs for both Java and PL/SQL, code samples, and documentation that address many of the design, development, and deployment issues encountered while creating global applications.

The GDK mainly consists of two parts: GDK for Java and GDK for PL/SQL. GDK for Java provides globalization support to Java applications. GDK for PL/SQL provides globalization support to the PL/SQL programming environment. The features offered in GDK for Java and GDK for PL/SQL are not identical.

The GDK for Java provides a J2EE application framework and Java APIs to develop global Internet applications using the best globalization practices and features designed by Oracle. It reduces the complexities and simplifies the code required to develop global Java applications.

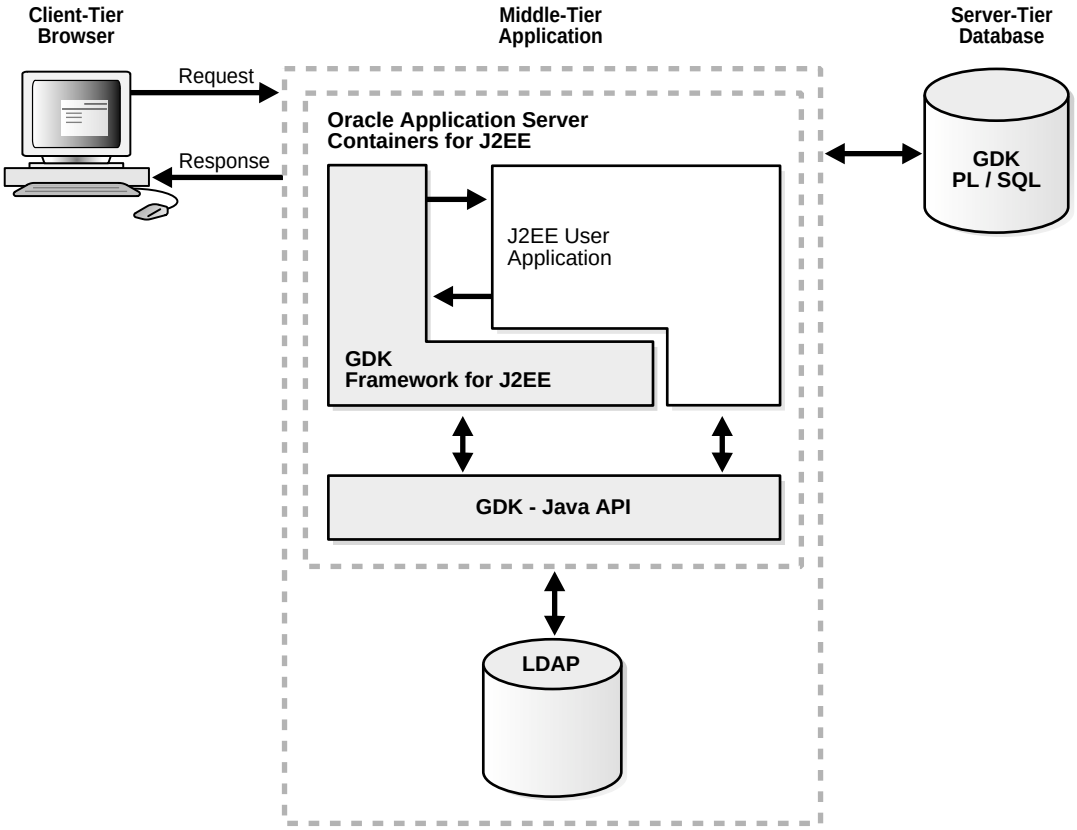
GDK for Java complements the existing globalization features in J2EE. Although the J2EE platform already provides a strong foundation for building global applications, its globalization functionalities and behaviors can be quite different from Oracle functionalities. GDK for Java provides synchronization of locale-sensitive behaviors between the middle-tier Java application and the database server.

GDK for PL/SQL contains a suite of PL/SQL packages that provide additional globalization functionalities for applications written in PL/SQL.

Figure 3–1 shows the major components of the GDK and how they relate to each other. User applications run on the J2EE container of Oracle Application Server in the middle tier. GDK provides the application framework that the J2EE application uses to

simplify coding to support globalization. Both the framework and the application call the GDK Java API to perform locale-sensitive tasks. GDK for PL/SQL offers PL/SQL packages that help to resolve globalization issues specific to the PL/SQL environment.

Figure 3–1 GDK Components



The functionalities offered by GDK for Java can be divided into two categories:

- The GDK application framework for J2EE provides the globalization framework for building J2EE-based Internet application. The framework encapsulates the complexity of globalization programming, such as determining user locale, maintaining locale persistency, and processing locale information. It consists of a set of Java classes through which applications can gain access to the framework. These associated Java classes enable applications to code against the framework so that globalization behaviors can be extended declaratively.
- The GDK Java API offers development support in Java applications and provides consistent globalization operations as provided in Oracle database servers. The API is accessible and is independent of the GDK framework so that standalone Java applications and J2EE applications not based on the GDK framework are able to access the individual features offered by the Java API. The features provided in the Java API include data and number formatting, sorting and character set handling in the same way as the Oracle database.

Note: The GDK Java API is certified with JDK versions 1.3 and later with the following exception: The character set conversion classes depend on the `java.nio.charset` package, which is available in JDK 1.4 and later.

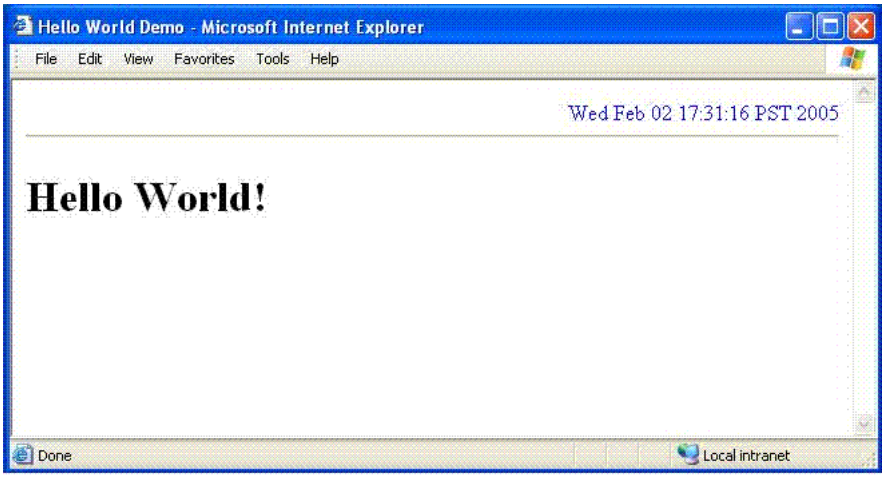
GDK for Java is contained in two files: `orai18n.jar` and `orai18n-lcsd.jar`. The files are shipped with Oracle Application Server. GDK is a pure Java library that runs on every platform. The Oracle client parameters `NLS_LANG` and `ORACLE_HOME` are not required.

GDK Quick Start

This section explains how to modify a monolingual application to be a global, multilingual application using GDK. The subsequent sections in this chapter provide detailed information on using GDK.

Figure 3–2 shows a monolingual Web application page.

Figure 3–2 Original HelloWorld Web Page



The initial, non-GDK HelloWorld Web application simply prints a "Hello World!" message, along with the current date and time in the top right hand corner of the page. The following code shows the original HelloWorld JSP source code for the preceding image.

Example 3–1 HelloWorld JSP Page Code

```
<%@ page contentType="text/html;charset=windows-1252"%>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
    <title>Hello World Demo</title>
  </head>
  <body>
  <div style="color: blue;" align="right">
    <%= new java.util.Date(System.currentTimeMillis()) %>
  </div>
  <hr/>
  <h1>Hello World!</h1>
  </body>
</html>
```

The following code example shows the corresponding Web application descriptor file for the HelloWorld message.

Example 3–2 HelloWorld web.xml Code

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <description>web.xml file for the monolingual Hello World</description>
  <session-config>
    <session-timeout>35</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>html</extension>
    <mime-type>text/html</mime-type>
  </mime-mapping>
  <mime-mapping>
    <extension>txt</extension>
    <mime-type>text/plain</mime-type>
  </mime-mapping>
</web-app>
```

The HelloWorld JSP code in Example 3–1 is only for English-speaking users. Some of the problems with this code are as follows:

- There is no locale determination based on user preference or browser setting.
- The title and the heading are included in the code.
- The date and time value is not localized based on any locale preference.
- The character encoding is for Latin-1.

The GDK framework can be integrated into the HelloWorld code to make it a global, multilingual application. The preceding code can be modified to include the following features:

- Automatic locale negotiation to detect the user's browser locale and serve the client with localized HTML pages. The supported application locales are configured in the GDK configuration file.
- Locale selection list to map the supported application locales. The list can have application locale display names which are the name of the country representing the locale. The list will be included on the Web page so users can select a different locale.
- GDK framework and API for globalization support for the HelloWorld JSP. This involves selecting display strings in a locale-sensitive manner and formatting the date and time value.

Modifying the HelloWorld Application

This section explains how to modify the HelloWorld application to support globalization. The application will be modified to support three locales, Simplified Chinese (zh-CN), Swiss German (de-CH), and American English (en-US). The following rules will be used for the languages:

- If the client locale supports one of these languages, then that language will be used for the application.
- If the client locale does not support one of these languages, then American English will be used for the application.

In addition, the user will be able to change the language by selecting a supported locales from the locale selection list. The following tasks describe how to modify the application:

- Task 1: Enable the Hello World Application to use the GDK Framework
- Task 2: Configure the GDK Framework for Hello World
- Task 3: Enable the JSP or Java Servlet
- Task 4: Create the Locale Selection List
- Task 5: Build the Application

Task 1: Enable the Hello World Application to use the GDK Framework

In this task, the GDK filter and a listener are configured in the Web application deployment descriptor file, `web.xml`. This allows the GDK framework to be used with the HelloWorld application. Example 3–3 shows the GDK-enabled `web.xml` file.

Example 3–3 The GDK-enabled web.xml File

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <description>web.xml file for Hello World</description>
   <!-- Enable the application to use the GDK Application Framework.-->
  <filter>
    <filter-name>GDKFilter</filter-name>
    <filter-class>oracle.i18n.servlet.filter.ServletFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>GDKFilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
  </filter-mapping>

  <listener>
    <listener-class>oracle.i18n.servlet.listener.ContextListener</listener-class>
  </listener>

  <session-config>
    <session-timeout>35</session-timeout>
  </session-config>
  <mime-mapping>
    <extension>html</extension>
    <mime-type>text/html</mime-type>
  </mime-mapping>
  <mime-mapping>
    <extension>txt</extension>
    <mime-type>text/plain</mime-type>
  </mime-mapping>
</web-app>
```

The following tags were added to the file:

- `<filter>`

  The filter name is GDKFilter, and the filter class is `oracle.i18n.servlet.filter.ServletFilter`.

- `<filter-mapping>`

The GDKFilter is specified in the tag, as well as the URL pattern.

- `<listener>`

  The listener class is `oracle.i18n.servlet.listener.ContextListener`. The default GDK listener is configured to instantiate GDK ApplicationContext, which controls application scope operations for the framework.

Task 2: Configure the GDK Framework for Hello World

The GDK application framework is configured with the application configuration file `gdkapp.xml`. The configuration file is located in the same directory as the `web.xml` file. Example 3–4 shows the `gdkapp.xml` file.

Example 3–4 GDK Configuration File gdkapp.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<gdkapp xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="gdkapp.xsd">

  <!-- The Hello World GDK Configuration -->
  <page-charset default="yes">UTF-8</page-charset>

  <!-- The supported application locales for the Hello World Application -->

  <application-locales>
    <locale>de-CH</locale>
    <locale default="yes">en-US</locale>
    <locale>zh-CN</locale>
  </application-locales>

  <locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>

<locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
  </locale-determine-rule>

  <message-bundles>
    <resource-bundle name="default">com.oracle.demo.Messages</resource-bundle>
  </message-bundles>
</gdkapp>
```

The file must be configured for J2EE applications. The following tags are used in the file:

- `<page-charset>`

  The page encoding tag specifies the character set used for HTTP requests and responses. The UTF-8 encoding is used as the default because many languages can be represented by this encoding.

- `<application-locales>`

  Configuring the application locales in the `gdkapp.xml` file makes a central place to define locales. This makes it easier to add and remove locales without changing source code. The locale list can be retrieved using the GDK API call `ApplicationContext.getSupportedLocales`.

- `<locale-determine-rule>`

  The language of the initial page is determined by the language setting of the browser. The user can override this language by choosing from the list. The

`locale-determine-rule` is used by GDK to first try the Accept-Language HTTP header as the source of the locale. If the user selects a locale from the list, then the JSP posts a locale parameter value containing the selected locale. The GDK then sends a response with the contents in the selected language.

- `<message-bundles>`

  The message resource bundles allow an application access to localized static content that may be displayed on a Web page. The GDK framework configuration file allows an application to define a default resource bundle for translated text for various languages. In the HelloWorld example, the localized string messages are stored in the Java ListResourceBundle bundle named `Messages`. The `Messages` bundle consists of base resources for the application which are in the default locale. Two more resource bundles provide the Chinese and German translations. These resource bundles are named `Messages_zh_CN.java` and `Messages_de.java` respectively. The HelloWorld application will select the right translation for "Hello World!" from the resource bundle based on the locale determined by the GDK framework. The `<message-bundles>` tag is used to configure the resource bundles that the application will use.

Task 3: Enable the JSP or Java Servlet

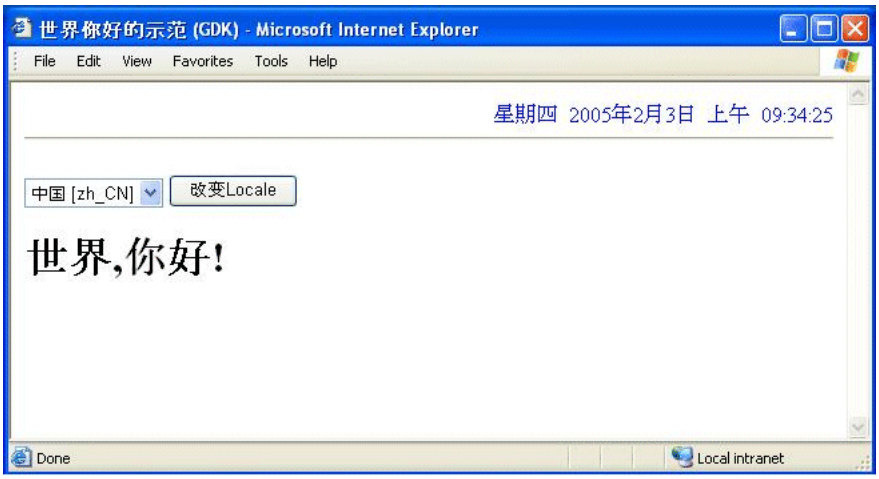
JSPs and Java servlets must be enabled to use the GDK API. Example 3–5 shows a JSP that has been modified to use the GDK API and services. This JSP can accommodate any language and locale.

Example 3–5 Enabled HelloWorld JSP

```
. . .
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title><%= localizer.getMessage("helloWorldTitle") %></title>
  </head>

  <body>
  <div style="color: blue;" align="right">
    <% Date currDate= new Date(System.currentTimeMillis()); %>
    <%=localizer.formatDateTime(currDate, OraDateFormat.LONG)%>
  </div>
  <hr/>

  <div align="left">
  <form>
    <select name="locale" size="1">
      <%= getCountryDropDown(request)%>
    </select>
    <input  type="submit" value="<%= localizer.getMessage("changeLocale") %>">
  </input>
  </form>
  </div>
  <h1><%= localizer.getMessage("helloWorld") %></h1>
  </body>
</html>
```

Figure 3–3 shows the HelloWorld application that has been configured with the zh-CN locale as the primary locale for the browser preference. The HelloWorld string and page title are displayed in Simplified Chinese. In addition, the date is formatted in the zh-CN locale convention. This example allows the user to override the locale from the locale selection list.

Figure 3–3 HelloWorld Localized for the zh-CN Locale



When the locale changes or is initialized using the HTTP Request Accept-Language header or the locale selection list, the GUI behaves appropriately for that locale. This means the date and time value in the upper right corner is localized properly. In addition, the strings are localized and displayed on the HelloWorld page.

The GDK Java Localizer class provides capabilities to localize the contents of a Web page based on the automatic detection of the locale by the GDK framework.

The following code retrieves an instance of the localizer based on the current HTTPServletRequest object. In addition, several imports are declared for use of the GDK API within the JSP page. The localizer retrieves localized strings in a locale-sensitive manner with fallback behavior, and formats the date and time.

```
<%@page contentType="text/html;charset=UTF-8"%>
<%@page import="java.util.*, oracle.i18n.servlet.*" %>
<%@page import="oracle.i18n.util.*, oracle.i18n.text.*" %>

<%
    Localizer localizer = ServletHelper.getLocalizerInstance(request);
%>
```

The following code retrieves the current date and time value stored in the currDate variable. The value is formatted by the localizer formatDateTime method. The OraDateFormat.LONG parameter in the formatDateTime method instructs the localizer to format the date using the locale's long formatting style. If the locale of the incoming request is changed to a different locale with the locale selection list, then the date and time value will be formatted according to the conventions of the new locale. No code changes need to be made to support newly-introduced locales.

```
div style="color: blue;" align="right">

    <% Date currDate= new Date(System.currentTimeMillis()); %>
    <%=localizer.formatDateTime(currDate, OraDateFormat.LONG)%>
  </div>
```

The HelloWorld JSP can be reused for any locale because the HelloWorld string and title are selected in a locale-sensitive manner. The translated strings are selected from a resource bundle.

The GDK uses the OraResourceBundle class for implementing the resource bundle fallback behavior. The following code shows how the Localizer picks the HelloWorld message from the resource bundle.

The default application resource bundle Messages is declared in the `gdkapp.xml` file. The localizer uses the message resource bundle to pick the message and apply the locale-specific logic. For example, if the current locale for the incoming request is "de-CH", then the message will first be looked for in the messages_de_CH bundle. If it does not exist, then it will look up in the Messages_de resource bundle.

```
<h1><%= localizer.getMessage("helloWorld") %></h1>
```

Task 4: Create the Locale Selection List

The locale selection list is used to override the selected locale based on the HTTP Request Accept-Language header. The GDK framework checks the locale parameter passed in as part of the HTTP POST request as a value for the new locale. A locale selected with the locale selection list is posted as the locale parameter value. GDK uses this value for the request locale. All this happens implicitly within the GDK code.

The following code sample displays the locale selection list as an HTML select tag with the name locale. The submit tag causes the new value to be posted to the server. The GDK framework retrieves the correct selection.

```
<form>
    <select name="locale" size="1">
      <%= getCountryDropDown(request)%>
    </select>
    <input type="submit" value="<%= localizer.getMessage("changeLocale") %>">
    </input>
</form>
```

The locale selection list is constructed from the HTML code generated by the `getCountryDropDown` method. The method converts the configured application locales into localized country names.

A call is made to the ServletHelper class to get the ApplicationContext object associated with the current request. This object provides the globalization context for an application, which includes information such as supported locales and configuration information. The `getSupportedLocales` call retrieves the list of locales in the `gdkapp.xml` file. The configured application locale list is displayed as options of the HTML select. The `OraDisplayLocaleInfo` class is responsible for providing localization methods of locale-specific elements such as country and language names.

An instance of this class is created by passing in the current locale automatically determined by the GDK framework. GDK creates requests and response wrappers for HTTP request and responses. The request.getLocale() method returns the GDK determined locale based on the locale determination rules.

The `OraDsiplayLocaleInfo.getDisplayCountry` method retrieves the localized country names of the application locales. An HTML option list is created in the ddOptBuffer string buffer. The `getCountryDropDown` call returns a string containing the following HTML values:

```
<option value="en_US" selected>United States [en_US]</option>
<option value="zh_CN">China [zh_CN]</option>
<option value="de_CH">Switzerland [de_CH]</option>
```

In the preceding values, the en-US locale is selected for the locale. Country names are generated are based on the current locale.

Example 3–6 shows the code for constructing the locale selection list.

Example 3–6 Constructing the Locale Selection List

```
<%!
    public String getCountryDropDown(HttpServletRequest request)
    {
        StringBuffer ddOptBuffer=new StringBuffer();
        ApplicationContext ctx =
ServletHelper.getApplicationContextInstance(request);
        Locale[] appLocales = ctx.getSupportedLocales();
        Locale currentLocale = request.getLocale();

        if (currentLocale.getCountry().equals(""))
        {
             // Since the Country was not specified get the Default Locale
             // (with Country) from the GDK
             OraLocaleInfo oli = OraLocaleInfo.getInstance(currentLocale);
             currentLocale = oli.getLocale();
        }

        OraDisplayLocaleInfo odli =
OraDisplayLocaleInfo.getInstance(currentLocale);
        for (int i=0;i<appLocales.length; i++)
        {
            ddOptBuffer.append("<option value=\"" + appLocales[i] + "\"" +
            (appLocales[i].getLanguage().equals(currentLocale.getLanguage()) ? "
selected" : "") +
                    ">" + odli.getDisplayCountry(appLocales[i]) +
                " [" +  appLocales[i] + "]</option>\n");
        }

        return ddOptBuffer.toString();
    }
%>
```

Task 5: Build the Application

In order to build the application, the following files must be specified in the classpath:

- `orai18n.jar`
- `regexp.jar`

The `orai18n.jar` file contains the GDK framework and the API. The `regexp.jar` file contains the regular expression library. The GDK API also has locale determination capabilities. The classes are supplied by the `ora18n-lcsd.jar` file.

GDK Application Configuration File

The GDK application configuration file dictates the behavior and properties of the GDK application framework and the application that is using it. It contains locale mapping tables and parameters for the configuration of the application. One configuration file is required for each application.

The `gdkapp.xml` application configuration file is an XML document. This file resides in the `./WEB-INF` directory of the J2EE environment of the application.

The following sections describe the contents and the properties of the application configuration file in detail:

- locale-charset-maps
- page-charset
- application-locales
- locale-determine-rule
- locale-parameter-name
- message-bundles
- url-rewrite-rule

locale-charset-maps

This tag enables applications to override the mapping from language to default character set provided by the GDK. This mapping is used when the `page-charset` is set to `AUTO-CHARSET`.

For example, for the `en` locale, the default GDK character set is WINDOWS-1252. However, if the application requires ISO-8859-1, this can be specified as follows:

```
<locale-charset-maps>
  <locale-charset>
    <locale>en</locale>
    <charset>ISO_8859-1</charset>
  </locale-charset>
</locale-charset-maps>
```

The locale name is comprised of the language code and the country code, and they should follow the ISO naming convention as defined in ISO 639 and ISO 3166, respectively. The character set name follows the Internet Assigned Numbers Authority (IANA) convention.

Optionally, the user-agent parameter can be specified in the mapping table to distinguish different clients.

```
<locale-charset>
  <locale>en,de</locale>
  <user-agent>^Mozilla/4.0</user-agent>
  <charset>ISO-8859-1</charset>
</locale-charset>
```

The preceding code shows that if the `user-agent` value in the HTTP header starts with `Mozilla/4.0` (which indicates older version of Web clients) for English (`en`) and German (`de`) locales, then the GDK sets the character set to ISO-8859-1.

Multiple locales can be specified in a comma-delimited list.

page-charset

This tag defines the character set of the application pages. If this is explicitly set to a given character set, then all pages use this character set. The character set name must follow the IANA character set convention.

```
<page-charset>UTF-8</page-charset>
```

However, if the `page-charset` is set to `AUTO-CHARSET`, then the character set is based on the default character set of the current user locale. The default character set is

derived from the locale to character set mapping table specified in the application configuration file.

If the character set mapping table in the application configuration file is not available, then the character set is based on the default locale name to IANA character set mapping table in the GDK. Default mappings are derived from `OraLocaleInfo` class.

See Also: "Handling Non-ASCII Input and Output in the GDK Framework" on page 3-24

application-locales

This tag defines a list of the locales supported by the application.

```
<application-locales>
  <locale default="yes">en-US</locale>
  <locale>de</locale>
  <locale>zh-CN</locale>
</application-locales>
```

If the language component is specified with the * country code, then all locale names with this language code qualify. For example, if `de-*` (the language code for German) is defined as one of the application locales, then this supports `de-AT` (German-Austria), `de` (German-Germany), `de-LU` (German-Luxembourg), `de-CH` (German-Switzerland), and even irregular locale combination such as `de-CN` (German-China). However, the application can be restricted to support a predefined set of locales.

It is recommended to set one of the application locales as the default application locale (by specifying `default="yes"`) so that it can be used as a fall back locale for customers who are connecting to the application with an unsupported locale.

locale-determine-rule

This tag defines the order in which the preferred user locale is determined. The locale sources should be specified based on the scenario in the application. The following scenarios describe how the locale-determine-rule is used with the GDK framework:

- Scenario 1: The GDK framework uses the accept language at all times.

  ```
  <locale-determine-rule>
     <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage
     </locale-source>
  </locale-determine-rule>
  ```

- Scenario 2: The GDK framework uses the accept language. After the user specifies the locale, the locale is used for further operations.

  ```
  <locale-determine-rule>
       <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
       <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage
       </locale-source>
  </locale-determine-rule>
  ```

- Scenario 3: The GDK framework uses the accept language. After the user is authenticated, the GDK framework uses the database locale source. The database locale source is cached until the user logs out. After the user logs out, the accept language is used again.

  ```
  <locale-determine-rule>
  ```

```
      <db-locale-source
         data-source-name="jdbc/OracleCoreDS"
         locale-source-table="customer"
         user-column="customer_email"
         user-key="userid"
         language-column="nls_language"
         territory-column="nls_territory"
         timezone-column="timezone">
      oracle.i18n.servlet.localesource.DBLocaleSource</db-locale-source>
      <locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage
      </locale-source>
</locale-determine-rule>
```

Scenario 3 includes the predefined database locale source, `DBLocaleSource`. It enables the user profile information to be specified in the configuration file without writing a custom database locale source. In the example, `customer` is the user profile table. The table has the following columns:

- `customer_email` for the unique e-mail address
- `nls_language` for the preferred language
- `nls_territory` for the Oracle name of the preferred territory
- `timezone` for the customer timezone

The `user-key` is a mandatory attribute that specifies the attribute name used to pass the user ID from the application to the GDK framework.

- Scenario 4: The GDK framework uses the accept language in the first page. When the user inputs a locale, it is cached and used until the user logs into the application. After the user is authenticated, the GDK framework uses the database locale source. The database locale source is cached until the user logs out. After the user logs out, the accept language is used or the user input is used, if the user inputs a locale.

  ```
  <locale-determine-rule>
       <locale-source>demo.DatabaseLocaleSource</locale-source>
       <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
       <locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage
       </locale-source>
  </locale-determine-rule>
  ```

  Scenario 4 uses the custom database locale source. If the user profile schema is complex, such as user profile information separated into multiple tables, then the custom locale source should be provided by the application. Examples of custom locale sources can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

locale-parameter-name

This tag defines the name of the locale parameters that are used in the user input so that the current user locale can be passed between requests.

```
<locale-parameter-name>
  <timezone>ti</timezone>
  <linguistic-sort>ls</linguistic-sort>
  <date-format>df</date-format>
</locale-parameter-name>
```

Table 3–1 shows the parameters used in the GDK framework.

Table 3–1 Locale Parameters Used in the GDK Framework

Default Parameter Name	Value
charset	Character set. For example, WE8ISO8859P15.
command	GDK command. For example, store for the update operation.
currency-format	Currency format. For example, L9G99G990D00.
date-format	Date format pattern mask. For example, DD_MON_RRRR.
date-time-format	Date and time format pattern mask. For example, DD-MON-RRRR HH24:MI:SS.
iso-currency	ISO 4217 currency code. For example, EUR for the Euro.
language	Oracle language name. For example, AMERICAN for American English.
linguistic-sort	Linguistic sort order name. For example, JAPANESE_M for Japanese multilingual sort.
locale	ISO locale where ISO 639 language code and ISO 3166 country code are connected with an underscore (_) or a hyphen (-). For example, zh_CN for Simplified Chinese used in China.
long-date-format	Long date format pattern mask. For example, DAY-YYYY-MM-DD.
long-date-time-format	Long date and time format pattern mask. For example, DAY YYYY-MM-DD HH12:MI:SS AM.
number-format	Number format. For example, 9G99G990D00.
territory	Oracle territory name. For example, SPAIN.
time-format	Time format pattern mask. For example, HH:MI:SS.
timezone	Timezone name. For example, American/Los_Angeles.
writing-direction	Writing direction string. LTR is used for left-to-right writing direction. RTL is used for right-to-left writing direction.

The parameter names are used in either the parameter in the HTML form or in the URL.

message-bundles

This tag defines the base class names of the resource bundles used in the application. The mapping is used in the Localizer.getMessage method for locating translated text in the resource bundles.

```
<message-bundles>
  <resource-bundle>Messages</resource-bundle>
  <resource-bundle name="newresource">NewMessages</resource-bundle>
</message-bundles>
```

If the name attribute is not specified or if it is specified as name="default" for the <resource-bundle> tag, then the corresponding resource bundle is used as the default message bundle. To support more than one resource bundle in an application, resource bundle names must be assigned to the nondefault resource bundles. The nondefault bundle names must be passed as a parameter of the getMessage method.

For example:

```
Localizer loc = ServletHelper.getLocalizerInstance(request);
String translatedMessage = loc.getMessage("Hello");
String translatedMessage2 = loc.getMessage("World", "newresource");
```

url-rewrite-rule

This tag is used to control the behavior of the URL rewrite operations. The rewriting rule is a regular expression.

```
<url-rewrite-rule fallback="no">
  <pattern>(.*)/([^/]+)$</pattern>
  <result>$1/$L/$2</result>
</url-rewrite-rule>
```

If the localized content for the requested locale is not available, then it is possible for the GDK framework to trigger the locale fallback mechanism by mapping it to the closest translation locale. By default, the fallback option is turned off. This can be turned on by specifying `fallback="yes"` in the expression.

For example, suppose an application has `fallback="yes"` and supports only the following translations `en`, `de`, and `ja`, with `en` as the default locale of the application. If the current application locale is `de-US`, then it falls back to `de`. If the user selects `zh-TW` as its application locale, then it falls back to `en`.

A fallback mechanism is often necessary if the number of supported application locales is greater than the number of the translation locales. This usually happens when multiple locales share one translation. One example is Spanish. The application may need to support multiple Spanish-speaking countries and not just Spain, with one set of translation files.

Multiple URL rewrite rules can be specified by assigning the name attribute to nondefault URL rewrite rules. To use the nondefault URL rewrite rules, the name must be passed as a parameter of the rewrite URL method. For example:

```
<img src="<%=ServletHelper.rewriteURL("images/welcome.gif", request) %>">
<img src="<%=ServletHelper.rewriteURL("US.gif", "flag", request) %>">
```

The first rule changes the `"images/welcome.gif"` URL to the localized welcome image file. The second rule named `"flag"` changes the `"US.gif"` URL to the user's country flag image file. The rule definition should be as follows:

```
<url-rewrite-rule fallback="yes">
  <pattern>(.*)/([^/]+)$</pattern>
  <result>$1/$L/$2</result>
</url-rewrite-rule>
<url-rewrite-rule name="flag">
  <pattern>US.gif</pattern>
  <result>$C.gif</result>
</url-rewrite-rule>
```

See Also: "Managing Localized Content in the GDK" on page 3-25

Example 3–7 shows an application configuration file with the following application properties:

- The application supports the following locales:
  - Arabic (`ar`)
  - Greek (`el`)
  - English (`en`)

  - German (de)
  - French (fr)
  - Japanese (ja)
  - Simplified Chinese for China (zh-CN)
- English is the default application locale.
- The page character set for the ja locale is always UTF-8.
- The page character set for the en and de locales when using an Internet Explorer client is WINDOWS-1252.
- The page character set for the en, de, and fr locales on other Web browser clients is ISO-8859-1.
- The page character sets for all other locales are the default character set for the locale.
- The user locale is determined in order by user input locale, then the Accept-Language tag.
- The localized contents are stored in their appropriate language subfolders. The folder names are derived from the ISO 639 language code. The folders are located in the root directory of the application. For example, the Japanese file for /shop/welcome.jpg is stored in /ja/shop/welcome.jpg.

Example 3–7 GDK Application Configuration File

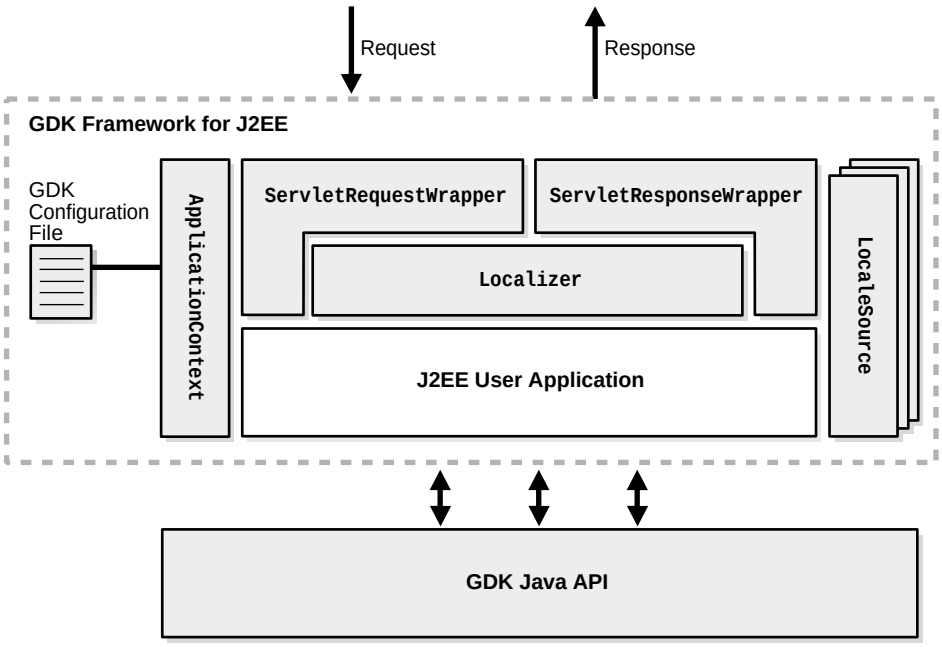
```
<?xml version="1.0" encoding="utf-8"?>
<gdkapp
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="gdkapp.xsd">
  <!-- Language to Character set mapping -->
  <locale-charset-maps>
    <locale-charset>
      <locale>ja</locale>
      <charset>UTF-8</charset>
    </locale-charset>
    <locale-charset>
      <locale>en,de</locale>
      <user-agent>^Mozilla\/[0-9\. ]+\(compatible; MSIE [^;]+; \)</user-agent>
      <charset>WINDOWS-1252</charset>
    </locale-charset>
    <locale-charset>
      <locale>en,de,fr</locale>
      <charset>ISO-8859-1</charset>
    </locale-charset>
  </locale-charset-maps>

  <!-- Application Configurations -->
  <page-charset>AUTO-CHARSET</page-charset>
  <application-locales>
    <locale>ar</locale>
    <locale>de</locale>
    <locale>fr</locale>
    <locale>ja</locale>
    <locale>el</locale>
    <locale>zh-CN</locale>
    <locale default="yes">en</locale>
  </application-locales>
  <locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
    <locale-source>oracle.i18n.servlet.localesource.HttpAcceptLanguage</locale-source>
  </locale-determine-rule>
```

```
  <!-- URL rewriting rule -->
  <url-rewrite-rule fallback="no">
    <pattern>(.*)/([^/]+)$</pattern>
    <result>/$L/$1/$2</result>
  </url-rewrite-rule>
</gdkapp>
```

GDK Application Framework for J2EE

GDK for Java provides the globalization framework for middle-tier J2EE applications. The framework encapsulates the complexity of globalization programming, such as determining user locale, maintaining locale persistency, and processing locale information. This framework minimizes the effort required to make Internet applications global-ready. The GDK application framework is shown in Figure 3–4.

Figure 3–4 GDK Application Framework for J2EE



The main Java classes composing the framework are as follows:

- `ApplicationContext` provides the globalization context of an application. The context information includes the list of supported locales and the rule for determining user-preferred locale. The context information is obtained from the GDK application configuration file for the application.
- The set of `LocaleSource` classes can be plugged into the framework. Each `LocaleSource` class implements the `LocaleSource` interface to get the locale from the corresponding source. Oracle bundles several `LocaleSource` classes in GDK. For example, the `DBLocaleSource` class obtains the locale information of the current user from a database schema. You can also write a customized `LocaleSource` class by implementing the same `LocaleSource` interface and plugging it into the framework.
- `ServletRequestWrapper` and `ServletResponseWrapper` are the main classes of the GDK Servlet filter that transforms HTTP requests and HTTP responses. `ServletRequestWrapper` instantiates a `Localizer` object for each HTTP request based on the information gathered from the

`ApplicationContext` and `LocaleSource` objects and ensures that forms parameters are handled properly. `ServletResponseWrapper` controls how HTTP response should be constructed.

- `Localizer` is the object that exposes the important functions sensitive to the current user locale and application context. It provides a centralized set of methods for you to call and make your applications behave appropriately to the current user locale and application context.
- The GDK Java API is always available for applications to enable finer control of globalization behavior.

The GDK application framework simplifies the coding required for your applications to support different locales. When you write a J2EE application according to the application framework, the application code is independent of what locales the application supports, and you control the globalization support in the application by defining it in the GDK application configuration file. There is no code change required when you add or remove a locale from the list of supported application locales.

The following list are some of the ways you can define globalization support in the GDK application configuration file:

- Add and remove a locale from the list of supported locales.
- Change the way the user locale is determined.
- Change the HTML page encoding of your application.
- Specify how to locate the translated resources.
- Implement a new `LocaleSource` object into the framework and use it to detect a user locale.

Making the GDK Framework Available to J2EE Applications

The behavior of the GDK application framework for J2EE is controlled by the GDK application configuration file, `gdkapp.xml`. The application configuration file allows developers to specify the behaviors of global applications in one centralized place. One application configuration file is required for each J2EE application using the GDK.

The `gdkapp.xml` file should be placed in the `./WEB-INF` directory of the J2EE environment of the application. It contains locale mapping tables, character sets of content files, and globalization parameters for the configuration of the application. The application administrator can modify the application configuration file to change the globalization behavior in the application, without changing the programs and recompiling them.

For a J2EE application to use the GDK application framework defined by the corresponding GDK application configuration file, the GDK Servlet filter and the GDK context listener must be defined in the `web.xml` file of the application. The `web.xml` file should be modified to include the following at the beginning of the file:

```
<web-app>
<!-- Add GDK filter that is called after the authentication -->

<filter>
    <filter-name>gdkfilter</filter-name>
    <filter-class>oracle.i18n.servlet.filter.ServletFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>gdkfilter</filter-name>
    <url-pattern>*.jsp</url-pattern>
```

```
</filter-mapping>

<!-- Include the GDK context listener -->

<listener>
<listener-class>oracle.i18n.servlet.listener.ContextListener</listener-class>
</listener>
</web-app>
```

Examples of the `gdkapp.xml` and `web.xml` files can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

The GDK application framework supports servlet container version 2.3 and later. It uses the servlet filter facility for transparent globalization operations such as determining the user locale and specifying the character set for content files. The `ContextListener` instantiates GDK application parameters described in the GDK application configuration file. The `ServletFilter` overrides the request and response objects with a GDK request (`ServletRequestWrapper`) and response (`ServletResponseWrapper`) objects, respectively.

If other application filters are used in the application to override the same methods, then the filter in the GDK framework may return incorrect results. For example, if `getLocale` returns `en_US`, but the result is overridden by other filters, then the result of the GDK locale detection mechanism is affected. Be aware of potential conflicts when using other filters together with the GDK framework.

See Also:

- "GDK Application Configuration File" on page 3-10
- *Oracle Globalization Development Kit Java API Reference* for more information about methods

Integrating Locale Sources into the GDK Framework

Determining the user's preferred locale is the first step in making an application global-ready. The locale detection offered by the J2EE application framework is primitive. It lacks a method that transparently retrieves the most appropriate user locale among locale sources. It provides locale detection by the HTTP language preference only, and it cannot support a multilevel locale fallback mechanism. The GDK application framework provides support for predefined locale sources to complement J2EE. In a Web application, several locale sources are available. Table 3–2 summarizes locale sources provided by the GDK.

Table 3–2 Locale Resources Provided by the GDK

Locale	Description
Application default locale	A locale defined in the GDK application configuration file. This locale is defined as the default locale for the application. Typically, this is used as a fallback locale when the other locale sources are not available.
HTTP language preference	Locales included in the HTTP protocol as a value of `Accept-Language`. This is set at the Web browser level. A locale fallback operation is required if the browser locale is not supported by the application.
User input locale	Locale specified by the user from a menu or a parameter in the HTTP protocol.

Table 3–2 (Cont.) Locale Resources Provided by the GDK

Locale	Description
User profile locale preference from database	Locale preference stored in the database as part of the user profiles.

The GDK application framework provides seamless support for predefined locale sources, such as user input locale, HTTP language preference, user profile locale preference in the database, and the application default locale. You can incorporate the locale sources to the framework by defining them under the `<locale-determine-rule>` tag in the GDK application configuration file as follows:

```
<locale-determine-rule>
    <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
    <locale-source>oracle.i18n.servlet.localesource.HTTPAcceptLanguage</locale-source>
</locale-determine-rule>
```

The GDK framework uses the locale source declaration order and determines whether a particular locale source is available. If it is available, then it is used as the source. If it is not available, then it tries to find the next available locale source for the list. In the preceding example, if the `UserInput` locale source is available, then it is used. If it is not, then the `HTTPAcceptLanguage` locale source will be used.

Custom locale sources, such as locale preference from an LDAP server, can be implemented and integrated into the GDK framework. You need to implement the `LocaleSource` interface and specify the corresponding implementation class under the `locale-determine-rule` tag in the same way as the predefined locale sources.

The `LocaleSource` implementation not only retrieves the locale information from the corresponding source to the framework but also updates the locale information to the corresponding source when the framework tells it to do so. Locale sources can be read-only or read / write, and they can be cacheable or noncacheable. The GDK framework initiates updates only to read / write locale sources and caches the locale information from cacheable locale sources. Examples of custom locale sources can be found in the `$ORACLE_HOME/nls/gdk/demo` directory.

See Also:

- "GDK Application Configuration File" on page 3-10 for information about the GDK multilevel locale fallback mechanism
- *Oracle Globalization Development Kit Java API Reference* for more information about implementing a `LocaleSource`

Getting the User Locale From the GDK Framework

The GDK offers automatic locale detection to determine the current locale of the user. For example, the following code retrieves the current user locale in Java. It uses a `Locale` object explicitly.

```
Locale loc = request.getLocale();
```

The `getLocale()` method returns the `Locale` that represents the current locale. This is similar to invoking the `HttpServletRequest.getLocale()` method in JSP or Java Servlet code. However, the logic in determining the user locale is different, because multiple locale sources are being considered in the GDK framework.

Alternatively, you can get a `Localizer` object that encapsulates the `Locale` object determined by the GDK framework.

```
Localizer localizer = ServletHelper.getLocalizerInstance(request);
Locale loc = localizer.getLocale();
```

> **See Also:** "Implementing Locale Awareness Using the GDK Localizer" on page 3-22 for information about the benefits of the `Localizer` object

The locale detection logic of the GDK framework depends on the locale sources defined in the GDK application configuration file. The names of the locale sources are registered in the application configuration file. The following example shows the locale determination rule section of the application configuration file. It indicates that the user-preferred locale can be determined from either the LDAP server or from the HTTP `Accept-Language` header. The `LDAPUserSchema` locale source class should be provided by the application. Note that all of the locale source classes have to be extended from the `LocaleSource` abstract class.

```
<locale-determine-rule>
    <locale-source>LDAPUserSchema</locale-source>
    <locale-source>oracle.i18n.localesource.HTTPAcceptLanguage</locale-source>
</locale-determine-rule>
```

For example, when the user is authenticated in the application and the user locale preference is stored in an LDAP server, then the `LDAPUserSchema` class connects to the LDAP server to retrieve the user locale preference. When the user is anonymous, the `HttpAcceptLanguage` class returns the language preference of the Web browser.

The cache is maintained for the duration of a HTTP session. If the locale source is obtained from the HTTP language preference, then the locale information is passed to the application in the HTTP `Accept-Language` header and not cached. This enables flexibility so that the locale preference can change between requests. The cache is available in the HTTP session.

The GDK framework has a method for the application to overwrite the locale preference information persistently stored in locale sources such as the LDAP server or the user profile table in the database. This method also resets the current locale information stored inside the cache for the current HTTP session. The following is an example of overwriting the preferred locale using the `store` command.

```
<input type="hidden"
name="<%=appctx.getParameterName(LocaleSource.Parameter.COMMAND)%>"
value="store">
```

To discard the current locale information stored inside the cache, the `clean` command can be specified as the input parameter. The following table shows the commands supported by the GDK:

Command	Functionality
`clean`	Discards the current locale information in the cache.
`store`	Updates user locale preferences in the available locale sources with the specified locale information. This command is ignored by the read-only locale sources.

Note that the GDK parameter names can be customized in the application configuration file to avoid name conflicts with other parameters used in the application.

Implementing Locale Awareness Using the GDK Localizer

The `Localizer` object obtained from the GDK application framework provides access to functions that are commonly used in building locale awareness in your applications. In addition, it provides functions to get information about the application context, such as the list of supported locales. The `Localizer` object simplifies and centralizes the code required to build consistent locale awareness behavior in your applications.

The `oracle.i18n.servlet` package contains the `Localizer` class. You can get the Localizer instance as follows:

```
Localizer lc = ServletHelper.getLocalizerInstance(request);
```

The `Localizer` object encapsulates the most commonly used locale-sensitive information determined by the GDK framework and exposes it as locale-sensitive methods. This object includes the following functionalities pertaining to the user locale:

- Format date in long and short formats
- Format numbers and currencies
- Get collation key value of a string
- Get locale data such as language, country and currency names
- Get locale data to be used for constructing user interface
- Get a translated message from resource bundles
- Get text formatting information such as writing direction
- Encode and decode URLs
- Get the common list of time zones and linguistic sorts

For example, when you want to display a date in your application, you may want to call the `Localizer.formatDate()` or `Localizer.formateDateTime()` methods. When you want to determine the writing direction of the current locale, you can call the `Localizer.getWritingDirection()` and `Localizer.getAlignment()` to determine the value used in the `<DIR>` tag and `<ALIGN>` tag respectively.

The `Localizer` object also exposes methods to enumerate the list of supported locales and their corresponding languages and countries in your applications.

The `Localizer` object actually makes use of the classes in the GDK Java API to accomplish its tasks. These classes includes, but are not limited to, the following:

- `OraDateFormat`
- `OraNumberFormat`
- `OraCollator`
- `OraLocaleInfo`
- `oracle.i18n.util.LocaleMapper`
- `oracle.i18n.net.URLEncoder`
- `oracle.i18n.net.URLDecoder`

The `Localizer` object simplifies the code you need to write for locale awareness. It maintains caches of the corresponding objects created from the GDK Java API so that the calling application does not need to maintain these objects for subsequent calls to the same objects. If you require more than the functionality the `Localizer` object can provide, then you can always call the corresponding methods in the GDK Java API directly.

> **See Also:** *Oracle Globalization Development Kit Java API Reference* for detailed information about the `Localizer` object

Defining the Supported Application Locales in the GDK

The number of locales and the names of the locales that an application needs to support are based on the business requirements of the application. The names of the locales that are supported by the application are registered in the application configuration file. The following example shows the application locales section of the application configuration file. It indicates that the application supports German (`de`), Japanese (`ja`), and English for the US (`en-US`), with English defined as the default fallback application locale. Note the locale names are based on the IANA convention.

```
<application-locales>
    <locale>de</locale>
    <locale>ja</locale>
    <locale default="yes">en-US</locale>
</application-locales>
```

When the GDK framework detects the user locale, it verifies whether the locale that is returned is one of the supported locales in the application configuration file. The verification algorithm is as follows:

1. Retrieve the list of supported application locales from the application configuration file.
2. Check whether the detected locale is included in the list. If it is included in the list, then use this locale as the current client's locale.
3. If there is a variant detected in the locale, then remove the variant and check whether the resulting locale is in the list. For example, the Java locale `de_DE_EURO` has a `EURO` variant. Remove the variant so that the resulting locale is `de_DE`.
4. If the locale includes a country code, then remove the country code and check whether the resulting locale is in the list. For example, the Java locale `de_DE` has a country code of `DE`. Remove the country code so that the resulting locale is `de`.
5. If the detected locale does not match any of the locales in the list, then use the default locale that is defined in the application configuration file as the client locale.

By performing steps 3 and 4, the application can support users with the same language requirements but with different locale settings than those defined in the application configuration file. For example, the GDK can support `de-AT` (the Austrian variant of German), `de-CH` (the Swiss variant of German), and `de-LU` (the Luxembourg variant of German) locales.

The locale fallback detection in the GDK framework is similar to that of the Java resource bundle, except that it is not affected by the default locale of the JVM. This exception occurs because the application default locale can be used during the GDK locale fallback operations.

If the application-locales section is omitted from the application configuration file, then the GDK assumes that the common locales, which can be returned by the `OraLocaleInfo.getCommonLocales` method, are supported by the application.

Handling Non-ASCII Input and Output in the GDK Framework

The character set or character encoding of an HTML page is important to a browser and an Internet application. The browser needs to interpret the information so it can use the correct fonts and character set mapping tables for displaying pages. The Internet applications need to know so they can safely process input data from an HTML form based on the specified encoding.

The page encoding can be translated as the character set used for the locale served by an Internet application. In order to correctly specify the page encoding for HTML pages without using the GDK framework, Internet applications must:

- Determine the desired page input data character set encoding for a given locale.
- Specify the corresponding encoding name for each HTTP request and HTTP response.

Applications using the GDK framework can ignore these steps. No application code change is required. The character set information is specified in the GDK application configuration file. At runtime, the GDK automatically sets the character sets for the request and response objects. The GDK framework does not support the scenario where the incoming character set is different from that of the outgoing character set.

The GDK application framework supports the following scenarios for setting the character sets of the HTML pages:

- A single local character set is dedicated to the whole application. This is appropriate for a monolingual Internet application. Depending on the properties of the character set, it may be able to support more than one language. For example, most Western European languages can be served by ISO-8859-1.
- Unicode UTF-8 is used for all contents regardless of the language. This is appropriate for a multilingual application that uses Unicode for deployment.
- The native character set for each language is used. For example, English contents are represented in ISO-8859-1, and Japanese contents are represented in Shift_JIS. This is appropriate for a multilingual Internet application that uses a default character set mapping for each locale. This is useful for applications that need to support different character sets based on the user locales. For example, for mobile applications that lack Unicode fonts or Internet browsers that cannot fully support Unicode, the character sets must be determined for each request.

The character set information is specified in the GDK application configuration file. The following is an example of setting UTF-8 as the character set for all the application pages.

```
<page-charset>UTF-8</page-charset>
```

The page character set information is used by the `ServletRequestWrapper` class, which sets the proper character set for the request object. It is also used by the `ContentType` HTTP header specified in the `ServletResponseWrapper` class for output when instantiated. If `page-charset` is set to `AUTO-CHARSET`, then the character set is assumed to be the default character set for the current user locale. Set `page-charset` to `AUTO-CHARSET` as follows:

```
<page-charset>AUTO-CHARSET</page-charset>
```

The default mappings are derived from the `LocaleMapper` class, which provides the default IANA character set for the locale name in the GDK Java API.

Table 3–3 lists the mappings between the common ISO locales and their IANA character sets.

Table 3–3 Mapping Between Common ISO Locales and IANA Character Sets

ISO Locale	IANA Character Set	NLS_LANGUAGE Value	NLS_TERRITORY Value
ar-SA	WINDOWS-1256	ARABIC	SAUDI ARABIA
de-DE	WINDOWS-1252	GERMAN	GERMANY
el	WINDOWS-1253	GREEK	GREECE
en-GB	WINDOWS-1252	ENGLISH	UNITED KINGDOM
en-US	WINDOWS-1252	AMERICAN	AMERICA
es-ES	WINDOWS-1252	SPANISH	SPAIN
fr	WINDOWS-1252	FRENCH	FRANCE
fr-CA	WINDOWS-1252	CANADIAN FRENCH	CANADA
it	WINDOWS-1252	ITALIAN	ITALY
iw	WINDOWS-1255	HEBREW	ISRAEL
ja	SHIFT_JIS	JAPANESE	JAPAN
ko	EUC-KR	KOREAN	KOREA
nl	WINDOWS-1252	DUTCH	THE NETHERLANDS
pt	WINDOWS-1252	PORTUGUESE	PORTUGAL
pt-BR	WINDOWS-1252	BRAZILIAN PORTUGUESE	BRAZIL
tr	WINDOWS-1254	TURKISH	TURKEY
zh	GBK	SIMPLIFIED CHINESE	CHINA
zh-TW	BIG5	TRADITIONAL CHINESE	TAIWAN

The locale to character set mapping in the GDK is also customizable. To override the default mapping defined in the GDK Java API, a locale-to-character-set mapping table can be specified in the application configuration file.

```
<locale-charset-maps>
   <locale-charset>
      <locale>ja</locale><charset>EUC-JP</charset>
   </locale-charset>
</locale-charset-maps>
```

The preceding example shows that for locale Japanese (`ja`), the GDK changes the default character set from SHIFT_JIS to EUC-JP.

See Also: "Oracle Locale Information in the GDK" on page 3-28

Managing Localized Content in the GDK

This section includes the following topics:

- Managing Localized Content in JSPs and Java Servlets

- Managing Localized Content in Static Files

Managing Localized Content in JSPs and Java Servlets

Resource bundles enable access to localized contents at runtime in Java 2 Platform, Standard Edition (J2SE). Translatable strings within Java servlets and Java Server Pages (JSPs) are externalized into Java resource bundles so these resource bundles can be translated independently into different languages. The translated resource bundles carry the same base class names as the English bundles, using the Java locale name as the suffix.

To retrieve translated data from the resource bundle, the `getBundle()` method must be invoked for every request.

```
<% Locale user_locale=request.getLocale();
   ResourceBundle rb=ResourceBundle.getBundle("resource",user_locale); %>
<%= rb.getString("Welcome") %>
```

The GDK framework simplifies the retrieval of text strings from the resource bundles. `Localizer.getMessage()` is a wrapper to the resource bundle.

```
<% Localizer.getMessage ("Welcome") %>
```

Instead of specifying the base class name as `getBundle()` in the application, you can specify the resource bundle in the application configuration file. The GDK will automatically instantiate a `ResourceBundle` object when a translated text string is requested.

```
<message-bundles>
  <resource-bundle name="default">resource</resource-bundle>
</message-bundles>
```

The preceding configuration file code declares a default resource bundle whose translated contents reside in the **resource** Java bundle class. Multiple resource bundles can be specified in the configuration file. To access a nondefault bundle, specify the `name` parameter in the `getMessage` method. The message bundle mechanism uses the `OraResourceBundle` GDK class for its implementation. This class provides the special locale fallback behaviors on top of the Java behaviors. The rules are as follows:

- If the given locale exactly matches the locale in the available resource bundles, then it will be used.
- If the resource bundle for Chinese in Singapore (`zh_SG`) is not found, then it will fallback to the resource bundle for Chinese in China (`zh_CN`) for Simplified Chinese translations.
- If the resource bundle for Chinese in Hong Kong (`zh_HK`) is not found, then it will fallback to the resource bundle for Chinese in Taiwan (`zh_TW`) for Traditional Chinese translations.
- If the resource bundle for Chinese in Macau (`zh_MO`) is not found, then it will fallback to the resource bundle for Chinese in Taiwan (`zh_TW`) for Traditional Chinese translations.
- If the resource bundles for any other Chinese locale (`zh_` and `zh`) is not found, then it will fallback to the resource bundle for Chinese in China (`zh_CN`) for Simplified Chinese translations.
- The default locale, which can be obtained by the `Locale.getDefault` method, will not be considered in the fallback operations.

The preceding rules provide the preferred languages for Hong Kong and Macau. The JDK fallback locale is Simplified Chinese (zh), whereas the users prefer Traditional Chinese (zh_TW).

For example, assume the default locale is `ja_JP` and the resource bundle for it is available. When the resource bundle for `es_MX` is requested and neither resource bundle for `es` or `es_MX` is provided, then the base resource bundle object that does not have the local suffix is returned.

The usage of the `OraResourceBundle` class is similar to the `java.util.ResourceBundle` class, but the `OraResourceBundle` call does not instantiate itself. Instead, the return value of the `getBundle` method is an instance of the subclass of the `java.util.ResourceBundle` class.

Managing Localized Content in Static Files

For an application that supports only one locale, the URL that has a suffix of `/index.html` typically takes the user to the starting page of the application.

In a global application, contents in different languages are usually stored separately, and it is common for them to be staged in different directories or with different file names based on the language or the country name. This information is then used to construct the URLs for localized content retrieval in the application.

The following code illustrates how to retrieve the French and Japanese versions of the index page. Their suffixes are as follows:

```
/fr/index.html
/ja/index.html
```

By using the `rewriteURL()` method of the ServletHelper class, the GDK framework handles the logic to locate the translated files from the corresponding language directories. The `ServletHelper.rewriteURL()` method rewrites a URL based on the rules specified in the application configuration file. This method is used to determine the correct location where the localized content is staged.

The following is an example of the JSP code:

```
<img src="<%="ServletHelper.rewriteURL("image/welcome.jpg", request)%>">
<a href="<%="ServletHelper.rewriteURL("html/welcome.html", request)%>">
```

The URL rewrite definitions are defined in the GDK application configuration file:

```
  <url-rewrite-rule fallback="yes">
    <pattern>(.*)/(a-zA-Z0-9_\]+.)$</pattern>
    <result>$1/$A/$2</result>
  </url-rewrite-rule>
```

The pattern section defined in the rewrite rule follows the regular expression conventions. The following special variables are for use with the `<result>` tag:

- `$L` represents the ISO 639 language code part of the current user locale
- `$C` represents the ISO 3166 country code
- `$A` represents the entire locale string, where the ISO 639 language code and ISO 3166 country code are connected with an underscore character (_)
- $1 to $9 represent the matched substrings

For example, if the current user locale is `ja`, then the URL for the `welcome.jpg` image file is rewritten as `image/ja/welcome.jpg`, and `welcome.html` is changed to `html/ja/welcome.html`.

Both `ServletHelper.rewriteURL()` and `Localizer.getMessage()` methods perform consistent locale fallback operations in the case where the translation files for the user locale are not available. For example, if the online help files are not available for the `es_MX` locale (Spanish for Mexico), but the `es` (Spanish for Spain) files are available, then the methods will select the Spanish translated files as the substitute.

GDK Java API

Java globalization functionalities and behaviors are not the same as those offered in the database. For example, J2SE supports a set of locales and character sets that are different from Oracle locales and character sets. This inconsistency can be confusing for users when their application contains data that is formatted based on 2 different conventions. For example, dates that are retrieved from the database are formatted using Oracle conventions, (such as number and date formatting and linguistic sort ordering), but the static application data is formatted using Java locale conventions. Java globalization functionalities can also be different depending on the version of the JDK that the application runs on.

The GDK Java API extends Oracle Application Server database globalization features to the middle tier. By enabling applications to perform globalization logic such as Oracle date and number formatting and linguistic sorting in the middle tier, the GDK Java API allows developers to eliminate expensive programming logic in the database, hence improving the overall application performance by reducing the database load in the database server and the unnecessary network traffic between the application tier and the database server.

The GDK Java API also offers advance globalization functionalities, such as language and character set detection, and the enumeration of common locale data for a territory or a language (for example, all time zones supported in Canada). These globalization features are not available in most programming platforms. Without the GDK Java API, developers must write business logic to handle them inside an application.

The following are the key functionalities of the GDK Java API:

- Oracle Locale Information in the GDK
- Oracle Locale Mapping in the GDK
- Oracle Character Set Conversion in the GDK
- Oracle Date, Number, and Monetary Formats in the GDK
- Oracle Binary and Linguistic Sorts in the GDK
- Oracle Language and Character Set Detection in the GDK
- Oracle Translated Locale and Time Zone Names in the GDK
- Using the GDK for E-mail Programs

Oracle Locale Information in the GDK

Oracle locale definitions, which include languages, territories, linguistic sorts, and character sets, are exposed in the GDK Java API. The naming convention Oracle uses may also be different from other vendors. Although many of these names and definitions follow industry standards, some are specific to Oracle, and tailored to meet special customer requirements.

The Oracle locale class, `OraLocaleInfo`, includes language, territory, and collator objects. It provides a method for applications to retrieve a collection of locale-related objects for a given locale. For example, a full list of the Oracle linguistic sorts available

in the GDK, the local time zones defined for a given territory, or the common languages used in a particular territory.

The following are examples using the `OraLocaleInfo` class:

```
// All Territories supported by GDK
String[] avterr = OraLocaleInfo.getAvailableTerritories();

// Local TimeZones for a given Territory
OraLocaleInfo oloc = OraLocaleInfo.getInstance("English", "Canada");
TimeZone[] loctz = oloc.getLocalTimeZones();
```

Oracle Locale Mapping in the GDK

The GDK Java API provides the `LocaleMapper` class. It maps equivalent locales and character sets between Java, IANA, ISO, and Oracle. A Java application may receive locale information from the client that is specified in the Oracle locale name or an IANA character set name. The Java application must be able to map to an equivalent Java locale or Java encoding before it can process the information correctly.

The following is an example of using the `LocaleMapper` class.

```
// Mapping from Java locale to Oracle language and Oracle territory

Locale locale = new Locale("it", "IT");
String oraLang = LocaleMapper.getOraLanguage(locale);
String oraTerr = LocaleMapper.getOraTerritory(locale);

// From Oracle language and Oracle territory to Java Locale

locale = LocaleMapper.getJavaLocale("AMERICAN","AMERICA");
locale = LocaleMapper.getJavaLocale("TRADITONAL CHINESE", "");

// From IANA & Java to Oracle Character set

String ocs1     = LocaleMapper.getOraCharacterSet(
                            LocaleMapper.IANA, "ISO-8859-1");
String ocs2     = LocaleMapper.getOraCharacterSet(
                            LocaleMapper.JAVA, "ISO8859_1");
```

The `LocaleMapper` class can also return the most commonly used e-mail character set for a specific locale on both Microsoft Windows and UNIX platforms. This is useful when developing Java applications that need to process e-mail messages.

See Also: "Using the GDK for E-mail Programs" on page 3-34

Oracle Character Set Conversion in the GDK

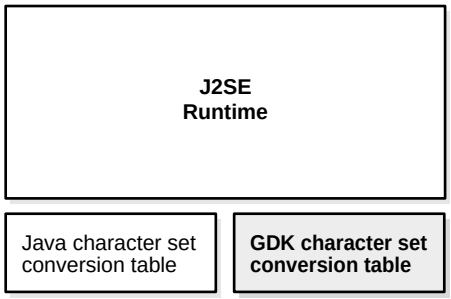
The GDK Java API contains a set of character set conversion class APIs that enable users to perform Oracle character set conversions. Although Java JDK is already equipped with classes that can perform conversions for many of the standard character sets, they do not support Oracle-specific character sets and Oracle user-defined character sets.

In JDK 1.4, J2SE introduced an interface for developers to extend Java character sets. The GDK Java API provides implicit support for Oracle character sets by using this plug-in feature. You can access the J2SE API to obtain behaviors specific for Oracle.

Figure 3–5 shows the GDK character set conversion tables are plugged into J2SE in the same way as the Java character set tables. With this pluggable framework of J2SE, the

Oracle character set conversions can be used in the same way as other Java character set conversions.

Figure 3–5 Oracle Character Set Plug-In



The `java.nio.charset` Java package is not available in JDK versions prior to version 1.4. You must install JDK 1.4 or later to use the Oracle character set plug-in feature.

The GDK character conversion classes support all Oracle character sets including user-defined characters sets. It can be used by Java applications to properly convert to and from Java internal character set, UTF-16.

Oracle character set names are proprietary. To avoid potential conflicts with Java's own character sets, all Oracle character set names have an `X-ORACLE-` prefix for all implicit usage through Java API.

The following is an example of Oracle character set conversion:

```
// Converts the Chinese character "three" from UCS2 to JA16SJIS

String str = "\u4e09";
byte[] barr = str.getBytes("x-oracle-JA16SJIS");
```

Like other Java character sets, the character set facility in `java.nio.charset.Charset` is also applicable to all of the Oracle character sets. For example, to check whether the specified character set is a superset of another character set, you can use the `Charset.contains` method as follows:

```
Charset cs1 = Charset.forName("x-oracle-US7ASCII");
Charset cs2 = Charset.forName("x-oracle-WE8WINDOWS1252");
// true if WE8WINDOWS1252 is the superset of US7ASCII, otherwise false.
boolean osc = cs2.contains(cs1);
```

For a Java application that is using the JDBC driver to communicate with the database, the JDBC driver provides the necessary character set conversion between the application and the database. Calling the GDK character set conversion methods explicitly within the application is not required. A Java application that interprets and generates text files based on Oracle character set encoding format is an example of using Oracle character set conversion classes.

Oracle Date, Number, and Monetary Formats in the GDK

The GDK Java API provides formatting classes that support date, number, and monetary formats using Oracle conventions for Java applications in the `oracle.i18n.text` package.

New locale formats introduced in Oracle Application Server, such as the short and long date, number, and monetary formats, are also provided in these format classes.

The following are examples of Oracle date, Oracle number, and Oracle monetary formatting:

```
// Obtain the current date and time in the default Oracle LONG format for
// the locale de_DE (German_Germany)

Locale locale = new Locale("de", "DE");
OraDateFormat odf =
  OraDateFormat.getDateTimeInstance(OraDateFormat.LONG, locale);

// Obtain the numeric value 1234567.89 using the default number format
// for the Locale en_IN (English_India)

locale = new Locale("en", "IN");
OraNumberFormat onf = OraNumberFormat.getNumberInstance(locale);
String nm = onf.format(new Double(1234567.89));

// Obtain the monetary value 1234567.89 using the default currency
// format for the Locale en_US (American_America)

locale = new Locale("en", "US");
onf = OraNumberFormat.getCurrencyInstance(locale);
nm = onf.format(new Double(1234567.89));
```

Oracle Binary and Linguistic Sorts in the GDK

Oracle provides support for binary, monolingual, and multilingual linguistic sorts in the database. In Oracle Application Server, these sorts have been expanded to provide case-insensitive and accent-insensitive sorting and searching capabilities inside the database. By using the `OraCollator` class, the GDK Java API enables Java applications to sort and search for information based on the latest Oracle binary and linguistic sorting features, including case-insensitive and accent-insensitive options.

Normalization can be an important part of sorting. The composition and decomposition of characters are based on the Unicode standard, so sorting also depends on the Unicode standard. Because each version of the JDK may support a different version of the Unicode Standard, the GDK provides an `OraNormalizer` class based on the Unicode 3.2 standard. It contains methods to perform composition.

The sorting order of a binary sort is based on the Oracle character set that is being used. Except for the UTFE character set, the binary sorts of all Oracle character sets are supported in the GDK Java API. The only linguistic sort that is not supported in the GDK Java API is JAPANESE, but a similar and more accurate sorting result can be achieved by using JAPANESE_M.

The following are examples of string comparisons and string sorting:

```
// compares strings using XGERMAN

private static String s1 = "abcSS";
private static String s2 = "abc\u00DF";

String cname = "XGERMAN";
OraCollator ocol = OraCollator.getInstance(cname);
int c = ocol.compare(s1, s2);


// sorts strings using GENERIC_M

private static String[] source =
  new String[]
```

```
  {
    "Hochgeschwindigkeitsdrucker",
    "Bildschirmfu\u00DF",
    "Skjermhengsel",
    "DIMM de Mem\u00F3ria",
    "M\u00F3dulo SDRAM com ECC",
  };


  cname = "GENERIC_M";
  ocol = OraCollator.getInstance(cname);
  List result = getCollationKeys(source, ocol);


private static List getCollationKeys(String[] source, OraCollator ocol)
{
  List karr = new ArrayList(source.length);
  for (int i = 0; i < source.length; ++i)
  {
    karr.add(ocol.getCollationKey(source[i]));
  }
  Collections.sort(karr); // sorting operation
  return karr;
}
```

Oracle Language and Character Set Detection in the GDK

The Oracle Language and Character Set Detection Java classes in the GDK Java API provide a high performance, statistically based engine for determining the character set and language for unspecified text. It can automatically identify language and character set pairs from throughout the world. With each text, the language and character set detection engine sets up a series of probabilities, each probability corresponding to a language and character set pair. The most probable pair statistically identifies the dominant language and character set.

The purity of the text submitted affects the accuracy of the language and character set detection. Only plain text strings are accepted, so any tagging needs to be stripped before hand. The ideal case is literary text with almost no foreign words or grammatical errors. Text strings that contain a mix of languages or character sets, or nonnatural language text like addresses, phone numbers, and programming language code may yield poor results.

The `LCSDetector` class can detect the language and character set of a byte array, a character array, a string, and an `InputStream` class. It can take the entire input for sampling or only portions of the input for sampling, when the length or both the offset and the length are supplied. For each input, up to three potential language and character set pairs can be returned by the `LCSDetector` class. They are always ranked in sequence, with the pair with the highest probability returned first.

The following are examples of using the `LCSDetector` class to enable language and character set detection:

```
// This example detects the character set of a plain text file "foo.txt" and
// then appends the detected ISO character set name to the name of the text file

LCSDetector         lcsd = new LCSDetector();
File             oldfile = new File("foo.txt");
FileInputStream      in = new FileInputStream(oldfile);
lcsd.detect(in);
String          charset = lcsd.getResult().getIANACharacterSet();
```

```
File            newfile = new File("foo."+charset+".txt");
oldfile.renameTo(newfile);

// This example shows how to use the LCSDector class to detect the language and
// character set of a byte array

int          offset = 0;
LCSDetector     led = new LCSDetector();
/* loop through the entire byte array */
while ( true )
{
    bytes_read = led.detect(byte_input, offset, 1024);
    if ( bytes_read == -1 )
        break;
    offset += bytes_read;
}
LCSDResultSet    res = led.getResult();

/* print the detection results with close ratios */
System.out.println("the best guess "  );
System.out.println("Language " + res.getOraLanguage() );
System.out.println("CharacterSet " + res.getOraCharacterSet() );
int     high_hit = res.getHiHitPairs();
if ( high_hit >= 2 )
{
        System.out.println("the second best guess  " );
        System.out.println("Language " + res.getOraLanguage(2) );
        System.out.println("CharacterSet " +res.getOraCharacterSet(2) );
}
if ( high_hit >= 3 )
{
         System.out.println("the third best guess  ");
         System.out.println("Language " + res.getOraLanguage(3) );
         System.out.println("CharacterSet " +res.getOraCharacterSet(3) );
}
```

Oracle Translated Locale and Time Zone Names in the GDK

All Oracle language names, territory names, character set names, linguistic sort names, and time zone names have been translated into 27 languages including English. They are readily available for inclusion in the user applications, and they provide consistency for the display names across user applications in different languages. `OraDisplayLocaleInfo` is a utility class that provides the translations of locale and attributes. The translated names are useful for presentation in user interface text and for selection boxes. For example, a native French speaker prefers to select from a list of time zones displayed in French than in English.

The following is an example of using `OraDisplayLocaleInfo` to return a list of time zones supported in Canada, using the French translation names:

```
OraLocaleInfo oloc = OraLocaleInfo.getInstance("CANADIAN FRENCH", "CANADA");
OraDisplayLocaleInfo odloc = OraDisplayLocaleInfo.getInstance(oloc);
TimeZone[] loctzs = oloc.getLocaleTimeZones();
String []  disptz = new string [loctzs.length];
for (int i=0; i<loctzs.length; ++i)
{
  disptz [i]= odloc.getDisplayTimeZone(loctzs[i]);
  ...
}
```

Using the GDK for E-mail Programs

You can use the GDK `LocaleMapper` class to retrieve the most commonly used e-mail character set. Call `LocaleMapper.getIANACharSetFromLocale`, passing in the locale object. The return value is an array of character set names. The first character set returned is the most commonly used e-mail character set.

The following is an example of sending an e-mail message containing Simplified Chinese data in GBK character set encoding:

```
import oracle.i18n.util.LocaleMapper;
import java.util.Date;
import java.util.Locale;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import javax.mail.internet.MimeUtility;
/**
 * Email send operation sample
 *
 * javac -classpath orai18n.jar:j2ee.jar EmailSampleText.java
 * java  -classpath .:orai18n.jar:j2ee.jar EmailSampleText
 */
public class EmailSampleText
{
  public static void main(String[] args)
  {
    send("localhost",                     // smtp host name
      "your.address@your-company.com",        // from email address
      "You",                             // from display email
      "somebody@some-company.com",           // to email address
      "Subject test zh CN",              // subject
      "Content •4E02 from Text email", // body
      new Locale("zh", "CN")             // user locale
    );
  }
  public static void send(String smtp, String fromEmail, String fromDispName,
    String toEmail, String subject, String content, Locale locale
  )
  {
    // get the list of common email character sets
    final String[] charset = LocaleMapper.getIANACharSetFromLocale(LocaleMapper.
EMAIL_WINDOWS,
locale
      );
    // pick the first one for the email encoding
    final String contentType = "text/plain; charset=" + charset[0];
    try
    {
      Properties props = System.getProperties();
      props.put("mail.smtp.host", smtp);
      //  here, set username / password if necessary
      Session session = Session.getDefaultInstance(props, null);
      MimeMessage mimeMessage = new MimeMessage(session);
      mimeMessage.setFrom(new InternetAddress(fromEmail, fromDispName,
          charset[0]
        )
      );
```

```
      mimeMessage.setRecipients(Message.RecipientType.TO, toEmail);
      mimeMessage.setSubject(MimeUtility.encodeText(subject, charset[0], "Q"));
      // body
      mimeMessage.setContent(content, contentType);
      mimeMessage.setHeader("Content-Type", contentType);
      mimeMessage.setHeader("Content-Transfer-Encoding", "8bit");
      mimeMessage.setSentDate(new Date());
      Transport.send(mimeMessage);
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

GDK for Java Supplied Packages and Classes

Oracle Globalization Services for Java contains the following packages:

- oracle.i18n.lcsd
- oracle.i18n.net
- oracle.i18n.servlet
- oracle.i18n.text
- oracle.i18n.util

See Also: *Oracle Globalization Development Kit Java API Reference*

oracle.i18n.lcsd

Package `oracle.i18n.lcsd` provides classes to automatically detect and recognize language and character set based on text input. Language is based on ISO, and encoding is based on IANA or Oracle character sets. It includes the following classes:

- `LCSDetector`: This class contains methods to automatically detect and recognize language and character set based on text input.
- `LCSDResultSet`: This class stores the result generated by `LCSDetector`. Methods in this class can be used to retrieve specific information from the result.

oracle.i18n.net

Package `oracle.i18n.net` provides Internet-related data conversions for globalization. It includes the following classes:

- `CharEntityReference`: A utility class to escape or unescape a string into character reference or entity reference form.
- `CharEntityReference.Form`: A form parameter class that specifies the escaped form.

oracle.i18n.servlet

Package `oracle.i18n.Servlet` enables JSP and JavaServlet to have automatic locale support and also returns the localized contents to the application. It includes the following classes:

- `ApplicationContext`: An application context class that governs application scope operation in the framework.
- `Localizer`: An all-in-one object class that enables access to the most commonly used globalization information.
- `ServletHelper`: A delegate class that bridges between Java servlets and globalization objects.

oracle.i18n.text

Package `oracle.i18n.text` provides general text data globalization support. It includes the following classes:

- `OraCollationKey`: A class which represents a `String` under certain rules of a specific `OraCollator` object.
- `OraCollator`: A class to perform locale-sensitive string comparison, including linguistic collation and binary sorting.
- `OraDateFormat`: An abstract class to do formatting and parsing between datetime and string locale. It supports Oracle datetime formatting behavior.
- `OraDecimalFormat`: A concrete class to do formatting and parsing between number and string locale. It supports Oracle number formatting behavior.
- `OraDecimalFormatSymbol`: A class to maintain Oracle format symbols used by Oracle number and currency formatting.
- `OraNumberFormat`: An abstract class to do formatting and parsing between number and string locale. It supports Oracle number formatting behavior.
- `OraSimpleDateFormat`: A concrete class to do formatting and parsing between datetime and string locale. It supports Oracle datetime formatting behavior.

oracle.i18n.util

Package `oracle.i18n.util` provides general utilities for globalization support. It includes the following classes:

- `LocaleMapper`: This class provides mappings between Oracle locale elements and the equivalent locale elements of other vendors and standards.
- `OraDisplayLocaleInfo`: A translation utility class that provides the translations of locale and attributes.
- `OraLocaleInfo`: An Oracle locale class that includes the language, territory, and collator objects.
- `OraSQLUtil`: An Oracle SQL utility class that includes some useful methods of dealing with SQL.

GDK for PL/SQL Supplied Packages

The GDK for PL/SQL includes the following PL/SQL packages:

- `UTL_I18N`
- `UTL_LMS`

`UTL_I18N` is a set of PL/SQL services that help developers to build global applications. The `UTL_I18N` PL/SQL package provides the following functions:

- String conversion functions for various datatypes
- Escape and unescape sequences for predefined characters and multibyte characters used by HTML and XML documents
- Functions that map between Oracle, IANA, ISO, and e-mail application character sets, languages, and territories
- A function that returns the Oracle character set name from an Oracle language name

`UTL_LMS` retrieves and formats error messages in different languages.

See Also: *Oracle9i Supplied PL/SQL Packages and Types Reference*

4

Implementing HTML Features

This chapter contains the following topics:

Implementing HTML Features for Global Applications

There are a variety of HTML features to enhance your global Internet applications. The following sections discuss some of the most important HTML features to consider when designing your global applications.

Formatting HTML Pages to Accommodate Text in Different Languages

Design the format of HTML pages according to the following guidelines:

- Allow table cells to resize themselves as the enclosed text expands, instead of hard-coding the widths of the cells. The following is an example of hard-coding the width of a cell:

  ```
  <TD WIDTH="50">
  ```

  If you must specify the widths of cells, then externalize the width values so that translators can adjust them with the translated text.

- It is good practice to provide a Cascading Style Sheets (CSS) for each locale or group of locales and use them to control HTML page rendering. Using a CSS isolates the locale-specific formatting information from HTML pages. Applications should dynamically generate CSS references in HTML pages corresponding to the user's locale so that the pages can be rendered with the corresponding locale-specific formats. Locale-specific information in the CSS file should include:
  - Font names and sizes
  - Alignments (for bidirectional language support only)
  - Direction of text (for bidirectional language support only)

- Do not specify fonts directly in the HTML pages because they may not contain glyphs for all languages that the application supports. Instead, each element should inherit fonts and font sizes from a class in a cascading style sheet (CSS).
- For bidirectional languages such as Arabic and Hebrew, the pages should have a `DIR` attribute in the `<HTML>` tag to indicate that the direction of the language displayed is from right to left. The `<HTML DIR="RTL">` tag causes all components of an HTML page to follow the direction of the HTML tag. To make direction settings seamless to developers, set the direction in the CSS file as follows:

  ```
  HTML{ direction:rtl }
  ```

  CSS level 2 introduced the direction property.
- Text alignment should be sensitive to the direction of the text. In HTML, LEFT and RIGHT are absolute alignments. When the direction of the text is from left to right, as in English, the alignment should be LEFT. When the direction of the text is from right to left, as in Hebrew, the alignment should be RIGHT.

Encoding HTML Pages

The encoding of an HTML page is important information for a browser and an Internet application. You can think of the **page encoding** as the character set used for the locale that an Internet application is serving. The browser needs to know about the page encoding so that it can use the correct fonts and character set mapping tables to display pages. Internet applications need to know about the HTML page encoding so they can process input data from an HTML form. To correctly specify the page encoding for HTML pages, Internet applications must:

- Choose a page encoding
- Encode HTML content in the desired encoding
- Correctly specify the HTML pages with the encoding name

Choosing an HTML Page Encoding for Monolingual Applications

The HTML page encoding is based on the user's locale. If the application is monolingual, it supports only one locale per instance. Therefore, you should encode HTML pages in the native encoding for that locale. The encoding should be equivalent to the Oracle character set specified by the `NLS_LANG` parameter in the Oracle HTTP Server configuration file.

Table 4–1 lists the Oracle character set names for the native encodings of the most commonly used locales, along with the corresponding Internet Assigned Numbers Authority (IANA) encoding names and Java encoding names. Use these character sets for monolingual applications.

Table 4–1 Native Encodings for Commonly Used Locales

Language	Oracle Character Set Name	IANA Encoding Name	Java Encoding Name
Arabic	AR8MSWIN1256	ISO-8859-6	ISO8859_6
Baltic	BLT8MSWIN1257	ISO-8859-4	ISO8859_4
Central European	EE8MSWIN1250	ISO-8859-2	ISO8859_2
Cyrillic	CL8MSWIN1251	ISO-8859-5	ISO8859_5

Table 4–1 (Cont.) Native Encodings for Commonly Used Locales

Language	Oracle Character Set Name	IANA Encoding Name	Java Encoding Name
Greek	EL8MSWIN1253	ISO-8859-7	ISO8859_7
Hebrew	IW8MSWIN1255	ISO-8859-8	ISO8859_8
Japanese	JA16SJIS	Shift_JIS	MS932
Korean	KO16MSWIN949	EUC-KR	MS949
Simplified Chinese	ZHS16GBK	GB2312	GBK
Thai	TH8TISASCII	TIS-620	TIS620
Traditional Chinese	ZHT16MSWIN950	Big5	MS950
Turkish	TR8MSWIN1254	ISO-8859-9	ISO8859_9
Universal	UTF8	UTF-8	UTF8
Western European	WE8MSWIN1252	ISO-8859-1	ISO8859_1

See Also: "Setting NLS_LANG for a Monolingual Application Architecture" in Chapter 6

Choosing an HTML Page Encoding for Multilingual Applications

Multilingual applications need to determine the encoding used for the current user's locale at runtime and map the locale to the encoding as shown in Table 4–1.

Instead of using different native encodings for different locales, you can use UTF-8 for all page encodings. Using the UTF-8 encoding not only simplifies the coding for multilingual applications but also supports multilingual content. In fact, if a multilingual Internet application is written in Perl, the best choice for the HTML page encoding is UTF-8 because these programming environments do not provide an intuitive and efficient way to convert HTML content from UTF-8 to the native encodings of various locales.

Specifying the Page Encoding for HTML Pages

The best practice for monolingual and multilingual applications is to specify the encoding of HTML pages returned to the client browser. The encoding of HTML pages can tell the browser to:

- Switch to the specified encoding
- Return user input in the specified encoding

The following sections explain how to specify the encoding of an HTML page:

- Specifying the Encoding in the HTTP Header
- Specifying the Encoding in the HTML Page Header

If you use both methods, then specifying the encoding in the HTTP header takes precedence.

Specifying the Encoding in the HTTP Header

Include the Content-Type HTTP header in the HTTP specification. It specifies the content type and character set. The most commonly used browsers, such as Netscape

4.0 or later and Internet Explorer 4.0 or later, correctly interpret this header. The Content-Type HTTP header has the following form:

```
Content-Type: text/plain; charset=iso-8859-4
```

The `charset` parameter specifies the encoding for the HTML page. The possible values for the `charset` parameter are the IANA names for the character encodings that the browser supports. Table 4–1 shows commonly used IANA names.

Specifying the Encoding in the HTML Page Header

Use this method primarily for static HTML pages. Specify the character encoding in the HTML header as follows:

```
<meta http-equiv="Content-Type" content="text/html;charset=utf-8">
```

The `charset` parameter specifies the encoding for the HTML page. The possible values for the charset parameter are the IANA names for the character encodings that the browser supports. Table 4–1 shows commonly used IANA names.

Specifying the Page Encoding in Java Servlets and Java Server Pages

For both monolingual and multilingual applications, you can specify the encoding of an HTML page in the Content-Type HTTP header in a Java Server Page (JSP) using the `contentType` page directive. For example:

```
<%@ page contentType="text/html; charset=utf-8" %>
```

This is the MIME type and character encoding that the JSP file uses for the response it sends to the client. You can use any MIME type or IANA character set name that is valid for the JSP container. The default MIME type is `text/html`, and the default character set is ISO-8859-1. In the example, the character set is set to UTF-8. The character set of the `contentType` page directive directs the JSP engine to encode the dynamic HTML page and set the HTTP Content-Type header with the specified character set.

For Java Servlets, you can call the `setContentType()` method of the Servlet API to specify a page encoding in the HTTP header. The following `doGet()` function shows how you should call this method:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{

    // generate the MIME type and character set header
    response.setContentType("text/html; charset=utf-8");
    ...
    // generate the HTML page
    Printwriter out = response.getWriter();
    out.println("<HTML>");
    ...
    out.println("</HTML>");
}
```

You should call the `setContentType()` method before the `getWriter()` method because the `getWriter()` method initializes an output stream writer that uses the character set specified by the `setContentType()` method call. Any HTML content written to the writer and eventually to a browser is encoded in the encoding specified by the `setContentType()` call.

See Also: "Handling Non-ASCII Input and Output in the GDK Framework" in Chapter 3

Specifying the Page Encoding in Oracle PL/SQL Server Pages

You can specify page encoding for PL/SQL front-end applications and Oracle PL/SQL Server Pages (PSP) in two ways:

- Specify the page encoding in the NLS_LANG parameter in the corresponding database access descriptor (DAD). Use this method for monolingual applications so you can change the page encoding without changing the application code to support a different locale.
- Specify the page encoding explicitly from within the PL/SQL procedures and PSP. A page encoding that is specified explicitly overwrites the page encoding inherited from the NLS_LANG character set. Use this method for multilingual applications so that they can use different page encodings for different locales at runtime.

The specified page encoding tells the mod_plsql module and the Web Toolkit to tag the corresponding charset parameter in the Content-Type header of an HTML page and to convert the page content to the corresponding character set.

See Also:

- "Configuring Transfer Mode for mod_plsql Runtime" in Chapter 6 for more information about configuring DADs
- *PL/SQL User's Guide and Reference* available in the Oracle Database 10*g* library on OTN at

  http://www.oracle.com/technology

Specifying the Page Encoding in PL/SQL for Monolingual Environments

In order for monolingual applications to take the page encoding from the NLS_LANG parameter, the Content-Type HTTP header should not specify a page encoding. For PL/SQL procedures, the call to mime_header() should be similar to the following:

```
owa_util.mime_header('text/html',false);
```

For PSP, the content type directive should be similar to the following:

```
<%@ page contentType="text/html"%>
```

If the page encoding is not specified in the mime_header() function call or the content type directive, then the Web Toolkit API uses the NLS_LANG character set as the page encoding by default, and converts HTML content to the NLS_LANG character set. Also, the Web Toolkit API automatically adds the default page encoding to the charset parameter of the Content-Type header.

Specifying the Page Encoding in PL/SQL for Multilingual Environments

You can specify page encoding in a PSP the same way that you specify it in a JSP page. The following directive tells the PSP compiler to generate code to set the page encoding in the HTTP Content-Type header for the page:

```
<%@ page contentType="text/html; charset=utf-8" %>
```

To specify the encoding in the Content-Type HTTP header for PL/SQL procedures, use the Web Toolkit API in the PL/SQL procedures. The Web Toolkit API consists of the OWA_UTL package, which allows you to specify the Content-Type header as follows:

```
owa_util.mime_header('text/html', false, 'utf-8')
```

You should call the `mime_header()` function in the context of the HTTP header. It generates the following Content-Type header in the HTTP response:

```
Content-Type: text/html; charset=utf-8
```

After you specify a page encoding, the Web Toolkit API converts HTML content to the specified page encoding.

Specifying the Page Encoding in Perl

For Perl scripts running in the `mod_perl` environment, specify the encoding for an HTML page in the HTTP Content-Type header as follows:

```
$page_encoding = 'utf-8';
$r->content_type("text/html; charset=$page_encoding");
$r->send_http_header;
return OK if $r->header_only;
```

See Also: *Oracle HTTP Server Administrator's Guide*

Specifying the Page Encoding in Perl for Monolingual Applications

For monolingual applications, the encoding of an HTML page should be equivalent to:

- The character set used for the POSIX locale on which a Perl script runs
- The Oracle character set specified by the `NLS_LANG` parameter if the Perl script accesses the database

Specifying the Page Encoding in Perl for Multilingual Applications

For multilingual applications, Perl scripts should run in an environment where:

- Both the `NLS_LANG` character set and the character set used for the POSIX locale are equivalent to UTF-8
- The UTF8 Perl pragma is used

  This pragma tells the Perl interpreter to encode identifiers and strings in the UTF-8 encoding.

  See Also: *Oracle HTTP Server Administrator's Guide*

This environment allows the scripts to process data in any language in UTF-8. The page encoding of the dynamic HTML pages generated from the scripts, however, could be different from UTF-8. If so, then use the `UNICODE::MAPUTF8` Perl module to convert data from UTF-8 to the page encoding.

See Also: `http://www.cpan.org` to download the `UNICODE::MAPUTF8` Perl module

The following example illustrates how to use the `UNICODE::MAPUTF8` Perl module to generate HTML pages in the Shift_JIS encoding:

```
use Unicode::MapUTF8 qw(from_utf8)
# This shows how the UTF8 Perl pragma is specified
# but is NOT required by the from_utf8 function.
use utf8;
...
```

```
$page_encoding = 'Shift_JIS';
$r->content_type("text/html; charset=$page_encoding");
$r->send_http_header;
return OK if $r->header_only;
...
#html_lines contains HTML content in UTF-8
print (from_utf8({ -string=>$html_lines, -charset=>$page_encoding}));
...
```

The `from_utf8()` function converts dynamic HTML content from UTF-8 to the character set specified in the `charset` argument.

Specifying the Page Encoding in Oracle Application Server Mobile Services Applications

The page encoding for a mobile services application is specified in the application in the same way as other Java or JSP Internet applications. The page encoding specifies the encoding of the Mobile XML generated by the application, and it should be consistently specified in the Mobile XML prolog and the HTTP Content-Type header. The `HelloGlobe.jsp` application illustrates how the page encoding for the Mobile XML prolog should be specified.

Example 4–1 HelloGlobe.jsp

```
<?xml version="1.0" encoding="UTF-8"?>   (1)
<%@ page contentType="text/vnd.oracle.mobilexml; charset=UTF-8"%>   (2)
<SimpleResult>
   <SimpleContainer>
      <SimpleForm title="Hello Globe"
                  target="HelloGlobeReply.jsp" method="POST">
         <SimpleFormItem name="UserName" title="Your Name:" />
      </SimpleForm>
   </SimpleContainer>
</SimpleResult>
```

In this example, line (1) sets the content encoding XML prolog, and line (2) sets the content encoding in the HTTP Content-Type header.

Oracle Application Server Wireless converts the Mobile XML into the page encoding supported by the target device from the encoding information specified in the XML prolog and the HTTP Content-Type header. It then renders the content in the markup language supported by the target device. If the encodings specified in the XML prolog and the HTTP Content-Type header are inconsistent, then the Oracle Application Server Wireless Mobile XML conversion will fail.

Specifying the Page Encoding in Oracle Web Cache Enabled Applications

When an edge side include (ESI) fragment is in a different page encoding from that of the corresponding ESI template, Oracle Web Cache converts the fragment to the page encoding of the template. This is to avoid cases where the content of a cached page is constructed in multiple page encodings. The character set conversion in Oracle Web Cache takes place only when both the template's and fragment's page encodings are known. Otherwise Oracle Web Cache assumes they are in the same page encoding, and therefore embeds the fragment into the template without converting the fragment.

Oracle Web Cache looks for the page encoding information only in the Content-Type header of an HTTP response. It does not look for the page encoding information within the content of the HTTP response.

To avoid losing information during the character set conversion of ESI fragments to ESI templates, applications should use a page encoding for ESI fragments that is a subset of the ESI template page encoding. There are two basic best practices for developers to consider:

- Use UTF-8 as the page encoding for ESI templates, since UTF-8 is a superset of all other non-Unicode page encodings.
- Use the same page encoding for ESI fragments and ESI templates. Character set conversion will not happen in this case.

Specifying the Page Encoding in Oracle Application Server Reports Services Applications

The page encodings that you use for different types of Reports Services applications depend on what type of report you are creating. This section discusses the page encoding options for Reports Services.

Specifying the Page Encoding in JSP Reports for the Web

You can specify the page encoding in JSP or HTML with the Web Source Editor in Reports Builder.

> **See Also:** "Specifying the Encoding in the HTML Page Header" and "Specifying the Page Encoding in Java Servlets and Java Server Pages" for more information

Specifying the Page Encoding in HTML for Oracle Application Server Reports Services

Specify the HTML page encoding in the page header. For example, to specify a Japanese character set, include the following tag in the page header:

```
<META http-equiv="Content-Type" content="text/html;charset=Shift_JIS">
```

> **See Also:** "Specifying the Encoding in the HTML Page Header"

Reports Builder puts this tag in your report using the `Before Report Value` and `Before Form Value` properties. The default values for these properties are similar to the following:

```
<html><head><meta http-equiv="Content-Type" content="text/html;charset=&Encoding"></head>
```

The IANA locale name that is equivalent to the `NLS_LANG` setting for Oracle Reports is assigned to `&Encoding` dynamically at runtime. Thus you do not need to modify your report or Oracle Reports settings to include the proper locale.

> **See Also:** Reports Builder online help for more information

Specifying the Page Encoding in XML for Oracle Reports

Generally, when using XML, you would specify the encoding for XML by including a statement similar to the following as the prolog at the first line in the XML output file:

```
<?xml version="1.0" encoding="Shift_JIS"?>
```

To set this prolog in your report, you can specify the `XML Prolog Value` property of your report in or use the `SRW.SET_XML_PROLOG` built-in. The default value for the `XML Prolog Value` property is:

```
<?xml version="1.0" encoding="&Encoding"?>
```

In this case, Reports translates the value set as the NLS_CHARACTERSET into what is expected in the XML specification.

Note: You can overwrite the mapping by adding entries to your REPORTS_NLS_XML_CHARSET. The syntax is:

old_name1=new_name1[;*old_name2=new_name2*][;*old_name3=new_name3*]...

Example:

```
ISO-8859-8=ISO-8859-8-1;CSEUCKR=EUC-KR;WINDOWS-949=EUC-KR;EUC-CN
=GBK;WINDOWS-936=GBK
```

See Also: Reports Builder online help for more information

Encoding URLs

If HTML pages contain URLs with embedded query strings, you must escape any non-ASCII bytes in the query strings in the %XX format, where XX is the hexadecimal representation of the binary value of the byte. For example, if an Internet application embeds a URL that points to a UTF-8 JSP containing the German name "Schloß," then the URL should be encoded as follows:

```
http://host.domain/actionpage.jsp?name=Schlo%c3%9f
```

In the preceding URL, `c3` and `9f` represent the binary value in hexadecimal of the ß character in the UTF-8 encoding.

To encode a URL, be sure to complete the following tasks:

1. Convert the URL into the encoding expected from the target object. This encoding is usually the same as the page encoding used in your application.
2. Escape non-ASCII bytes of the URL into the %XX format.

   Most programming environments provide APIs to encode and decode URLs. The following sections describe URL encoding in various environments:

   - Encoding URLs in Java
   - Encoding URLs in PL/SQL
   - Encoding URLs in Perl

Encoding URLs in Java

If you construct a URL in a JSP or Java Servlet, you must escape all 8-bit bytes using their hexadecimal values prefixed by a percent sign. The `URLEncoder.encode(String s, String enc)` function provided in JDK 1.4 and later enables you to escape the URL in a given HTML page encoding. You need to specify the proper Java encoding name that corresponds to the page encoding in the second argument. See Table 4–1 for the Java encoding names of some commonly used page encodings.

If you are using JDK 1.3, then only the `URLEncoder.encode(String s)` function is available. It only encodes a URL in the Java default encoding. To make this function work for URLs in any encoding, you must add code to escape any non-ASCII

characters in a URL into their hexadecimal representation, based on the encoding of your choice.

The following code shows an example of how to encode a URL based on the UTF-8 encoding:

```
String unreserved = new String("/\\-  _.!~*'()
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz 0123456789");
StringBuffer out = new StringBuffer(url.length());
for (int i = 0; i < url.length(); i++)
{
      int c = (int) url.charAt(i);
   if (unreserved.indexOf(c) != -1) {
        if (c == ' ') c = '+';
        out.append((char)c);
        continue;
   }
   byte [] ba;
   try {
        ba = url.substring(i, i+1).getBytes("UTF8");
   } catch (UnsupportedEncodingException e) {
        ba = url.getBytes();
   }
   for (int j=0; j < ba.length; j++)
   {
        out.append("%" + Long.toHexString((long)(ba[j]&0xff)).toUpperCase());
   }
}
String encodedUrl = out.toString();
```

Encoding URLs in PL/SQL

In Oracle Application Server, you can encode a URL in PL/SQL by calling the ESCAPE( ) function in the UTL_URL package. You can call the ESCAPE( ) function as follows:

```
encodedURL varchar2(100);
url varchar2(100);
charset varchar2(40);
...
encodedURL := UTL_URL.ESCAPE(url, FALSE, charset);
```

The `url` argument is the URL that you want to encode. The `charset` argument specifies the character encoding used for the encoded URL. Use a valid Oracle character set name for the `charset` argument. To encode a URL in the database character set, always specify the `charset` argument as NULL.

> **See Also:** Table 4–1 for a list of commonly used Oracle character set names

Encoding URLs in Perl

You can encode a URL in Perl by using the `escape_uri()` function of the `Apache::Util` module as follows:

```
use Apache::Util qw(escape_uri);
...
$escaped_url   = escape_uri( $url );
...
```

The `escape_uri()` function takes the bytes from the `$url` input argument and encodes them into the `%XX` format. If you want to encode a URL in a different character encoding, then you need to convert the URL to the target encoding before calling the `escape_uri()` function. Perl provides some modules for character conversion.

> **See Also:** `http://www.cpan.org` for Perl character conversion modules

Handling HTML Form Input

Applications generate HTML forms to get user input. For Netscape and Microsoft Internet Explorer browsers, the encoding of the input always corresponds to the encoding of the forms for both POST and GET requests. If the encoding of a form is UTF-8, then input text the browser returns is encoded in UTF-8. Internet applications can control the encoding of the form input by specifying the corresponding encoding in the HTML form that requests information.

How a browser passes input in a POST request is different from how it passes input in a GET request:

- For POST requests, the browser passes input as part of the request body. 8-bit data is allowed.
- For GET requests, the browser passes input as part of a URL as an embedded query string where every non-ASCII byte is encoded as `%XX`, where `XX` is the hexadecimal representation for the binary value of the byte.

HTML standards allow named and numbered entities. These special codes allow users to specify characters. For example, `&aelig;` and `&#230;` both refer to the character æ. Tables of these entities are available at

`http://www.w3.org/TR/REC-html40/sgml/entities.html`

Some browsers generate numbered or named entities for any input character that cannot be encoded in the encoding of an HTML form. For example, the Euro character and the character à (Unicode values 8364 and 224 respectively) cannot be encoded in Big5 encoding and are sent as `&#8364;` and `&agrave;` when the HTML encoding is Big5. However, the browser does not need to generate numbered or named entities if the page encoding of the HTML form is UTF-8 because all characters can be encoded in UTF-8. Internet applications that support page encoding other than UTF-8 need to be able to handle numbered and named entities.

Handling HTML Form Input in Java

In most JSP and Servlet containers the Servlet API implementation assumes that incoming form input is in ISO-8859-1 encoding. As a result, when the HttpServletRequest.getParameter( ) API is called, all embedded `%XX` data in the input text is decoded, the decoded input is converted from ISO-8859-1 to Unicode, and returned as a Java string. The Java string returned is incorrect if the encoding of the HTML form is not ISO-8859-1. However, you can work around this problem by converting the form input data. When a JSP or Java Servlet receives form input, it converts it back to the original form in bytes, and then converts the original form to a Java string based on the correct encoding.

The following code converts a Java string to the correct encoding. The Java string `real` is initialized to store the correct characters from a UTF-8 form:

```
String original = request.getParameter("name");
try
```

```
{
    String real = new String(original.getBytes("8859_1"),"UTF8");
}
catch (UnsupportedEncodingException e)
{
    String real = original;
}
```

In addition to Java encoding names, you can use IANA encoding names as aliases in Java functions.

> **See Also:** Table 4–1 for mapping between commonly used IANA and Java encoding names

OC4J implements Servlet API 2.3, from which you can get the correct input by setting the `CharEncoding` attribute of the HTTP request object before calling the `getParameter()` function. Use the following code:

```
request.setCharacterEncoding("UTF8");
String real = request.getParameter("name");
```

> **See Also:** "Handling Non-ASCII Input and Output in the GDK Framework" in Chapter 3

Handling HTML Form Input in PL/SQL

The browser passes form input to PL/SQL procedures as PL/SQL procedure arguments. When a browser issues a POST or a GET request, it first sends the form input to the `mod_plsql` module in the encoding of the requesting HTML form. The `mod_plsql` module then decodes all `%XX` escape sequences in the input to their actual binary representations. It then passes the input to the PL/SQL procedure serving the request.

You should construct PL/SQL arguments you use to accept form input with the VARCHAR2 datatype. Data in VARCHAR2 are always encoded in the database character set. For example, the following PL/SQL procedure accepts two parameters in VARCHAR2:

```
procedure test(name VARCHAR2, gender VARCHAR2)
begin
...
end;
```

By default, the `mod_plsql` module assumes that the arguments of a PL/SQL procedure are in VARCHAR2 datatype when it binds them. Using VARCHAR2 as the argument datatype means that the module uses Oracle Character Set Conversion facility provided in Oracle Callable Library to convert form input data properly from the NLS_LANG character set, which is also your page encoding, to the database character set. The corresponding DAD specifies the NLS_LANG character set. As a result, the arguments passed as VARCHAR2 should already be encoded in the database character set and be ready to use within the PL/SQL procedures.

Handling HTML Form Input in PL/SQL for Monolingual Applications

For monolingual application deployment, the NLS_LANG character set specified in the DAD is the same as the character set of the form input and the page encoding chosen for the locale. As a result, form input passed as VARCHAR2 arguments should be transparently converted to the database character set and ready for use.

Handling HTML Form Input in PL/SQL for Multilingual Applications

For multilingual application deployment, form input can be encoded in different character sets depending on the page encodings you choose for the corresponding locales. You cannot use Oracle Character Set Conversion facility because the character set of the form input is not always the same as the NLS_LANG character set. Relying on this conversion corrupts the input. To resolve this problem, disable Oracle Character Set Conversion facility by specifying the same NLS_LANG character set in the corresponding DAD as the database character set. Once you disable the conversion, PL/SQL procedures receive form input as VARCHAR2 arguments. You must convert the arguments from the form input encoding to the database character set before using them. You can use the following code to convert the argument from ISO-8859-1 character set to UTF-8:

```
procedure test(name VARCHAR2, gender VARCHAR2)
begin
   name := CONVERT(name, 'AMERICAN_AMERICA.UTF8',
                  AMERICAN_AMERICA.WE8MSWIN1252')
   gender := CONVERT(gender, 'AMERICAN_AMERICA.UTF8',
                  AMERICAN_AMERICA.WE8MSWIN1252')
...
end;
```

See Also: "Configuring the NLS_LANG Parameter" in Chapter 6

Handling HTML Form Input in Perl

In the Oracle HTTP Server `mod_perl` environment, GET requests pass input to a Perl script differently than POST requests. It is good practice to handle both types of requests in the script. The following code gets the input value of the `name` parameter from an HTML form:

```
my $r = shift;
my %params = $r->method eq 'POST' ? $r->content : $r->args ;
my $name = $params{'name'} ;
```

For multilingual Perl scripts, the page encoding of an HTML form may be different from the UTF-8 encoding used in the Perl scripts. In this case, input data should be converted from the page encoding to UTF-8 before being processed. The following example illustrates how the `Unicode::MapUTF8` Perl module converts strings from Shift_JIS to UTF-8:

```
use Unicode::MapUTF8 qw(to_utf8);
# This is to show how the UTF8 Perl pragma is specified,
# and is NOT required by the from_utf8 function.
use utf8;
...
my $page_encoding = 'Shift_JIS';
my $r = shift;
my %params = $r->method eq 'POST' ? $r->content : $r->args ;
my $name = to_utf8({-string=>$params{'name'}, -charset=>$page_encoding});
...
```

The `to_utf8()` function converts any input string from the specified encoding to UTF-8.

Handling Form Input in Oracle Application Server Mobile Services Applications

When a mobile service is registered to Oracle Application Server Wireless using the Wireless Tools administration tool, the Input Encoding parameter of the service must

be specified. Oracle Application Server Wireless encodes URL parameters using the encoding specified in the Input Encoding parameter of the service. The mobile service application should be written so that it uses the same encoding as the Input Encoding parameter to interpret input from the target mobile devices. The `HelloGlobeReply.jsp` example illustrates how to handle the response from the service `HelloGlobe.jsp`, which is described in Example 4–1.

Example 4–2 HelloGlobeReply.jsp

```
<%xml version="1.0" encoding="UTF-8"?>
<%@ page contentType="text/vnd.oracle.mobilexml; charset=UTF-8"%>
<%
   request.setCharacterEncoding("UTF-8");                              (1)
   String name = request.getParameter("UserName");
%>
<SimpleResult>
   <SimpleContainer>
      <SimpleText>
         <SimpleTextItem>Hello <%=name%> !</SimpleTextItem>
      </SimpleText>
   </SimpleContainer>
</SimpleResult>
```

In this example, line (1) specifies that parameters are encoded using UTF-8.

This assumes that the Input Encoding parameter is specified as UTF-8 when the Master Service of `HelloGlobe.jsp` is created. The mobile service application should specify the same encoding for all input parameters that are received from the target device.

Decoding HTTP Headers

In all HTTP headers specific to Oracle Application Server, any value containing non-ASCII characters is MIME encoded according to the RFC 2047 specification. The encoded headers must be properly decoded before being used in an application. Applications deployed on Oracle Application Server may receive these HTTP headers.

Decoding HTTP Headers from Oracle Single Sign-On

When applications are using Oracle Single Sign-On to authenticate a user, they need to decode the headers that Oracle Single Sign-On sends. The headers whose values may contain encoded non-ASCII characters include:

- `REMOTE_USER`
- `Osso-User-Dn`
- `Osso-Subscriber`
- `Osso-Subscriber-Dn`

For Java-based Web applications deployed on OC4J, the `REMOTE_USER` header is already interpreted in the `HTTPServletRequest.getRemoteUser()` method, and the `REMOTE_USER` header is removed from HTTP requests. For other types of Web applications, the `REMOTE_USER` header is present and should be properly decoded along with other headers.

To decode a header value, you may use the `javax.mail.internet.MimeUtility` package of the Java Mail API. See Example 4–3, "Decoding a User's Display Name" for an example of decoding.

For PL/SQL applications, you need to write your own code to decode these header values.

Decoding String-type Mobile Context Information Headers in Oracle Application Server Wireless Services

String-type mobile context information, such as Login User Name (`X-Oracle-User.name`), User Display Name (`X-Oracle-User.DisplayName`), and Address Line of the Location (`X-Oracle.User.Location.AddressLine1`) are MIME encoded in the HTTP headers. Applications must decode them after they are retrieved from the HTTP request. Example 4–3 shows how an JSP application may retrieve and decode the user's display name.

Example 4–3 Decoding a User's Display Name

```
<@ page import="java.io.*" %>
<@ page import="javax.mail.internet.MimeUtility" %>
<%
   String rawDisplayName = request.getHeader("X-Oracle-User.DisplayName");
   String displayName = null;
   try
   {
      displayName = MimeUtility.decodeText(rawDisplayName);
   }
   catch (UnsupportedEncodingException e)
   {
      // don't care
      displayName = rawDisplayName;
   }
%>
```

Organizing the Content of HTML Pages for Translation

You should translate the user interface (UI) and content presented in HTML pages. Translatable sources for the content of an HTML page belong to the following categories:

- Static files such as HTML, images, and cascading style sheets (CSS)
- Static UI strings stored as Java resource bundles used by Java Servlets and JSPs
- Static UI strings stored as POSIX message files used by C/C++ programs and Perl scripts
- Static UI strings stored as relational data in a database used by PL/SQL procedures and PL/SQL Server Pages
- Dynamic content such as product information stored in the database

This section contains the following topics:

- Organizing Translatable Dynamic Content in Application Schema

Translation Guidelines for HTML Page Content

When creating translatable content, developers should follow these translation guidelines:

- Externalize to resource files all static and translatable UI strings used in programs such as Java Servlets, Java Server Pages, Perl scripts, PL/SQL procedures, and PL/SQL Server Pages. These resource files can then be translated independent of program code.
- All dynamic text in an HTML page must be able to expand by at least 30% without overlapping adjacent objects to allow for text expansion that can result from translation. The HTML page should look acceptable after expanding strings by 30%.
- Avoid concatenating strings to form sentences at runtime. The concatenated translated strings might not have the same meaning as the original strings. Use the string formatting functions provided by different programming languages to substitute runtime values for placeholders.
- Avoid embedding text into images and graphics because they are often not easy to translate.
- JavaScript code must not include any translatable strings. JavaScript is hard to translate. Instead, applications should externalize translatable strings, if any, into resource files or message tables. Applications should construct JavaScript code at runtime and replace the dynamic text with text corresponding to the user's locale.
- Because translations are often not available in the initial release of an application, it is important to make the application work when the corresponding translation is not available by putting a fallback mechanism in the application. The fallback mechanism can be as simple as using English information or as complex as using the closest language available. For example, the `fr-CA` locale is French Canadian. The fallback for this language can be `fr` (French) or `en` (English). A simple way to find the closest possible language is to remove the territory part of the ISO locale name.The behavior of the fallback mechanism is up to the application.

Organizing Static Files for Translation

You should organize translatable HTML, images, and CSS files into different directories from non-translatable static files so that you can zip files under the locale-specific directory for translation. There are many possible ways to define the directory structure to hold these files. For example:

```
/docroot/images         - Non-translatable images
/docroot/html           - HTML pages common to all languages
/docroot/css            - Style sheets common to all languages
/docroot/lang         - Locale directory such as en, fr, ja, and so on.
/docroot/lang/images  - Images specific for lang
/docroot/lang/html    - HTML pages specific for lang
/docroot/lang/css     - Style sheets specific for lang
```

You can replace the `<lang>` placeholder with the ISO locale names. Based on the preceding structure, you must write a utility function called `getLocalizedURL ()`to take a URL as a parameter and look for the available language file from this structure. Whenever you reference an HTML, image, or CSS file in an HTML page, the Internet application should call this function to construct the path of the translated file

corresponding to the current locale and fall back appropriately if the translation does not exist. For example, if the path `/docroot/html/welcome.html` is passed to the `getLocalizedURL()` function and the current locale is `fr_CA`, then the function looks for the following files in the order shown:

```
/docroot/fr_CA/html/welcome.html
/docroot/fr/html/welcome.html
/docroot/en/html/welcome.html
/docroot/html/welcome.html
```

The function returns the first file that exists. This function always reverts to English when the translated version corresponding to the current locale does not exist.

For Internet applications that use UTF-8 as the page encoding, the encoding of the static HTML files should also be UTF-8. However, translators usually encode translated HTML files in the native encoding of the target language. To convert the translated HTML into UTF-8, you can use the JDK `native2ascii` utility shipped with Oracle Application Server.

For example, the following steps describe how to convert a Japanese HTML file encoded in Shift_JIS into UTF-8:

1. Replace the value of the `charset` parameter in the Content-Type HTML header in the `<meta>` tag with UTF-8.
2. Use the `native2ascii` utility to copy the Japanese HTML file to a new file called `japanese.unicode`:

   ```
   native2ascii -encoding MS932 japanese.html japanese.unicode
   ```

3. Use the `native2ascii` utility to convert the new file to Unicode:

   ```
   native2ascii -reverse -encoding UTF8 japanese.unicode japanese.html
   ```

   See Also: JDK documentation at `http://java.sun.com` for more information about the `native2ascii` utility

Organizing Translatable Static Strings for Java Servlets and Java Server Pages

You should externalize translatable strings within Java Servlets and JSPs into Java resource bundles so that these resource bundles can be translated independent of the Java code. After translation, the resource bundles carry the same base class names as the English bundles, but with the Java locale name as the suffix. You should place the bundles in the same directory as the English resource bundles for the Java resource bundle look-up mechanism to function properly.

See Also: JDK documentation at `http://java.sun.com` for more information about Java resource bundles

Some people may hesitate about externalizing JSP strings to resource bundles because it seems to defeat the purpose of using JSPs. There are two reasons for externalizing JSPs strings:

- Translating JSPs is error-prone because they consist of Java code that is not familiar to translators
- The translation process should be separated from the development process so that translation can take place in parallel to development on JSPs. This eliminates the huge effort of merging the translated JSPs with the most up-to-date JSPs that contain bug fixes to the embedded Java code.

You can use resource bundles in your Java programs by providing your own subclass of the `ResourceBundle` class. Additionally, Java provides two subclasses of the `ResourceBundle` abstract class: `ListResourceBundle` and `PropertyResourceBundle`. It is good practice to provide your implementation of the `ResourceBundle` class as a subclass of `ListResourceBundle`. The main reasons are:

- List resource bundles are essentially Java programs that must be compiled. Translation errors can be caught at compile time. Property resource bundles are text files read directly from Java. Translation errors can only be caught at runtime.
- Property resource bundles expose all string data in your Internet application to users. There are potential security and support issues for your application.

The following is an example of a list resource bundle:

```
import java.util.ListResourceBundle;
public class Resource extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents =
    {
       {"hello", "Hello World"},
       ...

    };
}
```

Translators usually translate list resource bundles in the native encoding of the target language. Japanese list resource bundles encoded in Shift_JIS cannot be compiled on an English system because the Java compiler expects source files that are encoded in ISO-8859-1. In order to build translated list resource bundles in a platform-independent manner, you need to run the JDK `native2ascii` utility to escape all non-ASCII characters to Unicode escape sequences in the `\uXXXX` format, where `XXXX` is the Unicode value in hexadecimal. For example:

```
native2ascii -encoding MS932 resource_ja.java resource_ja.tmp
```

Java provides a default fallback mechanism for resource bundles when translated resource bundles are not available. An application only needs to make sure that a base resource bundle without any locale suffix always exists in the same directory. The base resource bundle should contains strings in the fallback language. As an example, Java looks for a resource bundle in the following order when the `fr_CA` Java locale is specified to the `getBundle()` function:

```
resource_fr_CA
resource_fr
resource_en_US /* where en_US is the default Java locale */
resource_en
resource (base resource bundle)
```

Retrieving Strings in Monolingual Applications

At runtime, monolingual applications can get strings from a resource bundle of the default Java locale as follows:

```
ResourceBundle rb = ResourceBundle.getBundle("resource");
String helloStr = rb.getString("hello");
```

Retrieving Strings in Multilingual Applications

Because the user's locale is not fixed in multilingual applications, they should call the `getBundle()` method by explicitly specifying a Java locale object that corresponds to the user's locale. The Java locale object is called `user_locale` in the following example:

```
ResourceBundle rb = ResourceBundle.getBundle("resource", user_locale);
String helloStr = rb.getString("hello");
```

See Also: "Managing Localized Content in the GDK" in Chapter 3

Organizing Translatable Static Strings in C/C++ and Perl

For C/C++ programs and Perl scripts running on UNIX platforms, externalize static strings in C/C++ or Perl scripts to POSIX message files. For programs running on Microsoft Windows platforms, externalize static strings to message tables in a database because Microsoft Windows does not support POSIX message files.

See Also: "Organizing Translatable Static Strings in Message Tables"

Message files (with the `.po` file extension) associated with a POSIX locale are identified by their domain names. You need to compile them into binary objects with the `.mo` file extension and place them into the directory corresponding to the POSIX locale. The path name for the POSIX locale is implementation-specific. For example, the Solaris `msgfmt` utility compiles a French Canadian message file, `resource.po`, and places it into the `/usr/lib/locale/fr_CA/LC_MESSAGES` directory on Solaris.

See Also: Operating system documentation for `gettext`, `msgfmt`, and `xgettext`

The following is an example of a `resource.po` message file:

```
domain "resource"
msgid "hello"
msgstr "Hello World"
...
```

The encoding used for the message files must match the encoding used for the corresponding POSIX locale.

Instead of putting binary message files into an implementation-specific directory, you should put them into an application-specific directory and use the `binddomain()` function to associate a domain with a directory. The following piece of Perl script uses the `Locale::gettext` Perl module to get a string from a POSIX message file:

```
use Locale::gettext;
use POSIX;
...
setlocale( LC_ALL, "fr_CA" );
textdomain( "resource" );
binddomain( "resource", "/usr/local/share");
print gettext( "hello" );
```

The domain name for the resource file is `resource`, the ID of the string to be retrieved is `hello`, the translation to be used is French Canadian (`fr_ca`), and the directory for the `binary.mo` files is `/usr/local/share/fr_CA/LC_MESSAGES`.

> **See Also:** `http://www.cpan.org` to download the `Locale:gettext` Perl module

Organizing Translatable Static Strings in Message Tables

Message tables mainly store static translatable strings used by PL/SQL procedures and PSPs. You can also use them for some C/C++ programs and Perl scripts. The tables should have a language column to identify the language of static strings so that accessing applications can retrieve messages based on the user's locale. The table structure should be similar to the following:

```
CREATE TABLE messages
( msgid   NUMBER(5)
, langid  VARCHAR2(10)
, message VARCHAR2(4000)
);
```

The primary key for this table consists of the `msgid` and `langid` columns. One good choice for the values in these columns is the Oracle language abbreviations of corresponding locales. Using the Oracle language abbreviation allows applications to retrieve translated information transparently by issuing a query on the message table.

> **See Also:** *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library for a list of Oracle language abbreviations

To provide a fallback mechanism when the translation of a message is not available, create the following views on top of the message table defined in the previous example:

```
-- fallback language is English which is abbreviated as 'US'.
CREATE VIEW default_message_view AS
 SELECT msgid, message
 FROM messages
 WHERE langid = 'US';
/
-- create view for services, with fall-back mechanism
CREATE VIEW messages_view AS
SELECT d.msgid,
       CASE WHEN t.message IS NOT NULL
            THEN t.message
            ELSE d.message
       END AS message
FROM default_view d,
     translation  t
WHERE t.msgid (+) = d.msgid AND
      t.langid (+) = sys_context('USERENV', 'LANG');
```

Messages should be retrieved from the `messages_view` view that provides the logic for a fallback message in English by joining the `default_message_view` view with the `messages` table. The `sys_context()` SQL function returns the Oracle language abbreviation of the locale for the current database session. This locale should be initialized to the user's locale at the time when the session is created.

To retrieve a message, an application should use the following query:

```
SELECT message FROM message_view WHERE msgid = 'hello';
```

The NLS_LANGUAGE parameter of a database session defines the language of the message that the query retrieves. Note that there is no language information needed for the query with this message table schema.

In order to minimize the load to the database, you should set up all message tables and their associated views on an Oracle Application Server instance as a front end to the database where PL/SQL procedures and PSPs run.

See Also: "Managing Localized Content in the GDK" in Chapter 3

Organizing Translatable Dynamic Content in Application Schema

An application schema stores translatable dynamic information the application uses, such as product names and product descriptions. The following shows an example of a table that stores all the products of an Internet store. The translatable information for the table is the product name and the product description.

```
CREATE TABLE product_information
    ( product_id          NUMBER(6)
    , product_name        VARCHAR2(50)
    , product_description VARCHAR2(2000)
    , category_id         NUMBER(2)
    , warranty_period     INTERVAL YEAR TO MONTH
    , supplier_id         NUMBER(6)
    , product_status      VARCHAR2(20)
    , list_price          NUMBER(8,2)
    );
```

To store product names and product descriptions in different languages, create the following table so that the primary key consists of the `product_id` and `language_id` columns:

```
CREATE TABLE product_descriptions
    ( product_id             NUMBER(6)
    , language_id            VARCHAR2(3)
    , translated_name        NVARCHAR2(50)
    , translated_description NVARCHAR2(2000)
    );
```

Create a view on top of the tables to provide fallback when information is not available in the language that the user requests. For example:

```
CREATE VIEW product AS
SELECT i.product_id
,      d.language_id
,      CASE WHEN d.language_id IS NOT NULL
            THEN d.translated_name
            ELSE i.product_name
       END    AS product_name
,      i.category_id
,      CASE WHEN d.language_id IS NOT NULL
            THEN d.translated_description
            ELSE i.product_description
       END    AS product_description
,      i.warranty_period
,      i.supplier_id
,      i.product_status
,      i.list_price
FROM   product_information  i
,      product_descriptions d
WHERE  d.product_id  (+) = i.product_id
```

```
AND    d.language_id (+) = sys_context('USERENV','LANG');
```

This view performs an outer join on the `product_information` and `production_description` tables and selects the rows with the `language_id` equal to the Oracle language abbreviation of the current database session.

To retrieve a product name and product description from the product view, an application should use the following query:

```
SELECT product_name, product_description FROM product
      WHERE product_id = '1234';
```

This query retrieves the translated product name and production description corresponding to the value of the `NLS_LANGUAGE` session parameter. Note that you do not need to specify any language information in the query because the query uses `sys_context ('USERENV', 'LANG')`, which returns the session language.

5

Using a Centralized Database

This chapter contains the following topics:

Using a Centralized Database and Accessing the Database Server

A centralized Unicode database is a feature of both the monolingual approach and the multilingual approach to developing global Internet applications. Using a centralized database has the following advantages:

- It provides a complete view of your data. For example, you can query for the number of customers worldwide or the worldwide inventory level of a product.
- It is easier to manage a centralized database than several distributed databases.

The database character set should be Unicode. You can use Unicode to store and manipulate data in several languages. Unicode is a universal character set that defines characters in almost all languages in the world. Oracle databases can store Unicode data in one of the following encoding forms:

- UTF-8: Each character is 1 to 4 bytes long.
- UTF-16: Each character is either 2 or 4 bytes long.
- UTF-32: Each character is 4 bytes long.

See Also:

- "Configuring Oracle HTTP Server and OC4J for Global Deployment" in Chapter 6
- *Oracle Globalization Support Guide* in the Oracle Database Documentation Library

There are several methods by which Internet applications can access the database server through Oracle Application Server. Any Java-based Internet applications that use technologies such as Java Servlets, JSPs, and EJBs can use the Oracle JDBC drivers for database connectivity.

Because Java strings are always Unicode-encoded, JDBC transparently converts text data from the database character set to and from Unicode. Java Servlets and JSPs that interact with an Oracle database should ensure the following:

- The Java strings returned from the database are converted to the encoding of the HTML page being constructed
- Form inputs are converted from the encoding of the HTML form to Unicode before being used in calling the JDBC driver

For non-Java Internet applications that use programming technologies such as Perl, PL/SQL, and C/C++, text data retrieved from or inserted into a database are encoded in the character set specified by the NLS_LANG parameter. The character set used for the POSIX locale should match the NLS_LANG character set so that data from the database can be directly processed with the POSIX locale-sensitive functions in the applications.

For multilingual applications, the NLS_LANG character set and the page encoding should both be UTF-8 to avoid character set conversion and possible data loss.

See Also: Chapter 6, "Configuring Oracle Application Server for Global Deployment"

Using JDBC to Access the Database

Use the Oracle JDBC drivers provided in Oracle Application Server for Oracle database access when you use JSPs and Java Servlets. Oracle Application Server provides two client-side JDBC drivers that you can deploy with middle-tier applications:

- JDBC OCI driver, which requires the Oracle client library
- JDBC Thin driver, which is a pure Java driver

Oracle JDBC drivers transparently convert character data from the database character set to Unicode for the SQL CHAR data types and the SQL NCHAR data types. As a result of this transparent conversion, JSPs and Java Servlets calling Oracle JDBC drivers can bind and define database columns with Java strings and fetch data into Java strings from the result set of a SQL execution.

You can use a Java string to bind the NAME and ADDRESS columns of a customer table. Define the columns as VARCHAR2 and NVARCHAR2 data types, respectively. For example:

```
String cname = request.getParameter("cname")
String caddr = request.getParameter("caddress");
OraclePreparedStatement pstmt = conn.prepareStatement("insert into" +
        "CUSTOMERS (NAME, ADRESS) values (?, ?) ");
pstmt.setString(1, cname);
pstmt.setFormOfUse(2, OraclePreparedStatement.FORM_NCHAR);
pstmt.setString(2, caddr);
pstmt.execute();
```

To bind a Java string variable to the ADDRESS column defined as NVARCHAR2, you should call the `setFormOfUse()` method before the `setString()` method.

The Oracle JDBC drivers set the values for the NLS_LANGUAGE and NLS_TERRITORY session parameters to the values corresponding to the default Java locale when the database session was initialized. For monolingual applications, the Java default locale is configured to match the user's locale. Hence the database connection is always synchronized with the user's locale.

See Also: *Oracle Database JDBC Developer's Guide and Reference* in the Oracle Database Documentation Library

Using PL/SQL to Access the Database

PL/SQL procedures and PSPs use SQL to access data in the local Oracle database. They can also use SQL and database links to access data from a remote Oracle database.

For example, you can call the following PL/SQL procedure from the mod_plsql module. It inserts a record into a customer table with the customer name column defined as VARCHAR2 and the customer address column defined as NVARCHAR2:

```
procedure addcustomer( cname varchar2 default NULL, caddress nvarchar2 default
NULL) is
begin
   ....
   if (cname is not null) then
        caddr :=TO_NCHAR(address);
        insert into customers (name, address) values (cname, caddr);
        commit;
   end if;
end;
```

Note that Apache mod_plsql does not support NVARCHAR argument passing. As a result, PL/SQL procedures have to use VARCHAR2 for arguments and convert them to NVARCHAR explicitly before executing the INSERT statement.

The example uses static SQL to access the customer table. You can also use the DBMS_SQL PL/SQL package to access data in the database, using dynamic SQL.

See Also: *PL/SQL Packages and Types Reference 10g Release 1 (10.1)* in the Oracle Database Documentation Library

Using Perl to Access the Database

Perl scripts access Oracle databases using the DBI/DBD driver for Oracle. The DBI/DBD driver is part of Oracle Application Server. It calls Oracle Callable Interface (OCI) to access the databases. The data retrieved from or inserted into the databases is encoded in the NLS_LANG character set. Perl scripts should do the following:

- Initialize a POSIX locale with the locale specified in the LC_ALL environment variable
- Use a character set equivalent to the NLS_LANG character set

This allows you to process data retrieved from the databases with POSIX string manipulation functions.

The following code shows how to insert a row into a customer table in an Oracle database through the DBI/DBD driver.

```
Use Apache::DBI;
...
# Connect to the database
$constr = 'host=dlsun1304.us.oracle.com;sid=icachedb;port=1521' ;
$usr = 'system' ;
$pwd = 'manager' ;
$dbh = DBI->connect("dbi:Oracle:$constr", $usr, $pwd, {AutoCommit=>1} ) ||
       $r->print("Failed to connect to Oracle: " . DBI->errstr );
```

```
# prepare the statement
$sql = 'insert into customers (name, address) values (:n, :a)';
$sth = $dbh->prepare( $sql );
$sth->bind_param(':n' , $cname);
$sth->bind_param(':a', $caddress);
$sth->execute();
$sth->finish();
$dbh->disconnect();
```

If the target columns are of the SQL NCHAR data types, then you need to specify the form of use flag for each bind variable. For example, if the address column is of NVARCHAR2 datatype, then you need to add the `$sth->func()` function call before executing the SQL statement:

```
use DBD::Oracle qw(:ora_forms);
...
$sql = 'insert into customers (name, address) values (:n, :a)';
$sth = $dbh->prepare($sql);
$sth->bind_param(':n', $cname);
$sth->bind_param(':a', $caddress);
$sth->func( { ':a' => ORA_NCHAR }, 'set_form');
$sth->execute();
$sth->finish();
$dbh->disconnect();
```

To properly process UTF-8 data in a multilingual application, Perl scripts should do the following:

- Use a POSIX locale associated with the UTF-8 character set
- Use the UTF-8 Perl module to indicate that all strings in the Perl scripts are in UTF-8

Using C/C++ to Access the Database

C/C++ applications access the Oracle databases with OCI or Pro*C/C++. You can call OCI directly or use the Pro*C/C++ interface to retrieve and store Unicode data in a UTF-8 database and in SQL NCHAR data types.

Generally, data retrieved from and inserted into the database is encoded in the NLS_LANG character set. C/C++ programs should use the same character set for their POSIX locale as the NLS_LANG character set. Otherwise, the POSIX string functions cannot be used on the character data retrieved from the database, and the character data encoded in the POSIX locale may be corrupted when it is inserted into the database.

For multilingual applications, you may want to use the Unicode API provided in the OCI library instead of relying on the NLS_LANG character set. This alternative is good for applications written for platforms such as Microsoft Windows, which implement the `wchar_t` datatype using UTF-16 Unicode. Using the Unicode API for those platforms bypasses some unnecessary data conversions required when using the regular OCI API.

This section includes the following topics:

- Using the OCI API to Access the Database
- Using the Unicode API Provided with OCI to Access the Database
- Using Unicode Bind and Define in Pro*C/C++ to Access the Database

Note: OCI libraries are part of Oracle Application Server. You do not need to install the Oracle database client to use them.

Using the OCI API to Access the Database

The following example shows how to bind and define the VARCHAR2 and NVARCHAR2 columns of a customer table in C/C++. It uses OCI and is based on the NLS_LANG character set. Note that the `text` datatype is a macro for `unsigned char`.

```
text *sqlstmt= (text *)"SELECT name, address FROM customers
                        WHERE id = :cusid";
text cname[100];                        /* Customer Name */
text caddr[200];                        /* Customer Address */
text custid[10] = "9876";               /* Customer ID */
ub2 cform = SQLCS_NCHAR;                /* Form of Use for NCHAR types */
...
OCIStmtPrepare (stmthp, errhp, sqlstmt,
                (ub4)strlen ((char *)sqlstmt),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
/* Bind the custid buffer */
OCIBindByName(stmthp, &bnd1p, errhp, (text*)":custid",
              (sb4)strlen((char *)":custid"),
              (dvoid *) custid, sizeof(cust_id), SQLT_STR,
              (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0,(ub4 *)0, OCI_DEFAULT);

/* Define the cname buffer for VARCHAR */
OCIDefineByPos (stmthp, &dfn1p, errhp, (ub4)1, (dvoid *)cname,
                (sb4)sizeof(cname), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);

/* Define the caddr buffer for the address column in NVARCHAR2 */
OCIDefineByPos (stmthp, &dfn2p, errhp, (ub4)2, (dvoid *)caddr,
                (sb4)sizeof(caddr), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfn2p, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...
```

Using the Unicode API Provided with OCI to Access the Database

You can use the Unicode API that the OCI library provides for multilingual applications.

Turn on the Unicode API by specifying Unicode mode when you create an OCI environment handle. Any handle inherited from the OCI environment handle is set to Unicode mode automatically. By changing to Unicode mode, all text data arguments to the OCI functions are assumed to be in the Unicode text (`utext*`) datatype and in UTF-16 encoding. For binding and defining, the data buffers are assumed to be `utext` buffers in UTF-16 encoding.

The program code for the Unicode API is similar to the code for the non-Unicode OCI API, with the following exceptions:

- All `text` data types are changed to the `utext` datatype, which is a macro of the unsigned short datatype
- All literal strings are changed to Unicode literal strings

- All `strlen()` functions are changed to `wcslen()` functions to calculate the string length in number of Unicode characters instead of bytes

The following Microsoft Windows program shows how to do these tasks:

- Create an OCI environment handle with Unicode mode turned on
- Bind and define the name column in VARCHAR2 and the address column in NVARCHAR2 of the `customers` table

```
utext *sqlstmt= (text *)L"SELECT name, address FROM customers
                         WHERE id = :cusid";
utext cname[100];                     /* Customer Name */
utext caddr[200];                     /* Customer Address */
text custid[10] = "9876";             /* Customer ID */
ub1 cform = SQLCS_NCHAR;              /* Form of Use for NCHAR types */
...
/* Use Unicode OCI API by specifying UTF-16 mode */
status = OCIEnvCreate((OCIEnv **)&envhp, OCI_UTF16, (void *)0,
                      (void *(*) ()) 0, (void *(*) ()) 0, (void(*) ()) 0,
                      (size_t) 0, (void **)0);
...
OCIStmtPrepare (stmthp, errhp, sqlstmt,
                (ub4)wcslen ((char *)sqlstmt),
                (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));

/* Bind the custid buffer */
OCIBindByName(stmthp, &bnd1p, errhp, (constant text*) L":custid",
              (sb4)wcslen(L":custid"),
              (dvoid *) custid, sizeof(cust_id), SQLT_STR,
              (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0,
              (ub4) 0,(ub4 *)0, OCI_DEFAULT);

/* Define the cname buffer for the name column in VARCHAR2 */
OCIDefineByPos (stmthp, &dfn1p, errhp, (ub4)1, (dvoid *)cname,
                (sb4)sizeof(cname), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);

/* Define the caddr buffer for the address column in NVARCHAR2 */
OCIDefineByPos (stmthp, &dfn2p, errhp, (ub4)2, (dvoid *)caddr,
                (sb4)sizeof(caddr), SQLT_STR,
                (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet((void *) dfn2p, (ub4) OCI_HTYPE_DEFINE, (void *) &cform, (ub4) 0,
           (ub4)OCI_ATTR_CHARSET_FORM, errhp);
...
```

Using Unicode Bind and Define in Pro*C/C++ to Access the Database

You can use Unicode bind and define in Pro*C/C++ for multilingual applications.

Pro*C/C++ lets you specify UTF-16 Unicode buffers for bind and define operations. There are two ways to specify UTF-16 buffers in Pro*C/C++:

- Use the `utext` datatype, which is an alias for the unsigned short datatype in C/C++
- Use the `uvarchar` datatype provided by Pro*C/C++. It will be preprocessed to a `struct` with a length field and a `utext` buffer field.

```
struct uvarchar
{
   ub2 len;         /* length of arr */
```

```
   utext arr[1] ;   /* UTF-16 buffer */
};
typedef struct uvarchar uvarchar ;
```

In the following example, there are two host variables: `cname` and `caddr`. The `cname` host variable is declared as a `utext` buffer containing 100 UTF-16 code units (unsigned short) for the customer name column in the VARCHAR2 datatype. The `caddr` host variable is declared as a `uvarchar` buffer containing 50 UCS2 characters for the customer address column in the NVARCHAR2 datatype. The `len` and `arr` fields are accessible as fields of a `struct`.

```
#include <sqlca.h>
#include <sqlucs2.h>

main()
{
   ...
   /* Change to STRING datatype:    */
   EXEC ORACLE OPTION (CHAR_MAP=STRING) ;
   utext cname[100] ;              /* unsigned short type */
   uvarchar caddr[200] ;           /* Pro*C/C++ uvarchar type */
   ...
   EXEC SQL SELECT name, address INTO :cname, :caddr FROM customers;
   /* cname is NULL-terminated */
   wprintf(L"ENAME = %s, ADDRESS = %.*s\n", cname, caddr.len, caddr.arr);
   ...
}
```

6

Configuring Oracle Application Server for Global Deployment

When developing and deploying global Internet applications with Oracle Application Server, you need to consider the following tasks:

- Installing Oracle Application Server for Global Deployment
- Configuring Oracle HTTP Server and OC4J for Global Deployment
- Configuring Oracle Application Server Portal for Global Deployment
- Configuring Oracle Application Server Wireless for Global Deployment
- Configuring Oracle Application Server Forms Services for Global Deployment
- Configuring OracleAS Reports Services for Global Deployment
- Configuring Oracle Business Intelligence Discoverer for Global Deployment
- Configuring a Centralized Unicode-enabled Database to Support Global Deployment

Installing Oracle Application Server for Global Deployment

In addition to the schemas of the infrastructure components, such as Oracle Internet Directory and Distributed Configuration Management (DCM), the Oracle Application Server Infrastructure database stores data pertaining to many Oracle Application Server middle-tier components that are installed on top of it. These components include the following:

- Oracle Application Server Portal (OracleAS Portal)
- Oracle Application Server Forms Services (OracleAS Forms Services)
- Oracle Reports (OracleAS Reports Services)
- Oracle Application Server Wireless (OracleAS Wireless)
- Oracle Business Intelligence Discoverer (OracleBI Discoverer)

It is important to choose the correct database character set for the infrastructure database at installation time so all the dependent components are able to provide the same level of global support.

During the installation of the Oracle Application Server infrastructure database, you are prompted to choose the database character set you would like to use for the database. The default character set is AL32UTF8. There are two basic scenarios that will determine which choice is best for your environment:

- If your environment is intended to support multiple languages in a single global instance of the Oracle Application Server infrastructure, similar to Figure 1–2, then choose UTF-8 as the character set for the infrastructure database. Even if you only support a single language, such as English, you may choose UTF-8 as the database character set. The implications of choosing UTF-8 include, but are not limited to, the following:
  - Databases with UTF-8 as the database character set are slightly slower than those with single-byte character sets. The performance impact is due to UTF-8 being a multibyte character set and the increase in the number of character set conversions between the middle tier and the database.
  - Web pages served through `mod_plsql` must be encoded in UTF-8. However, there are some browsers, such as those for mobile devices, that may have problems supporting UTF-8. Products that deliver Web pages through `mod_plsql` include Oracle Single Sign-On and Oracle Application Server Portal.
- If your environment is intended to support a single language or a group of languages that share the same native character set, then you can choose the character set that is most commonly used for these languages as an alternative to UTF-8. For example, you can choose WE8MSWIN1252 if you are only interested in supporting Western European languages. You can choose JA16SJIS if you are interested in supporting Japanese and English.
  - If your environment will support Traditional Chinese, then select ZHT16MSWIN950 or ZHT16BIG5 as the character set.

Note: The character set ZHT32EUC does not support Oracle Application Server Portal.

During installation of any Oracle Application Server installation type, support for user-selected languages is automatically installed and configured. It includes the translation files and fonts being used in the product.

If the required fonts are not available after installation, then you can copy them from the Utilities CD-ROM included in the Oracle Application Server CD pack, or from

`http://metalink.oracle.com`

into the `$ORACLE_HOME/jdk/jre/lib/fonts` directory.

See Also: *Oracle Application Server Portal Developer's Guide* for information about creating multilingual portlets

Configuring Oracle HTTP Server and OC4J for Global Deployment

This section contains the following topics related to configuring Oracle HTTP Server for multilingual support:

- About Manually Editing HTTP Server and OC4J Configuration Files
- Configuring the NLS_LANG Parameter
- Configuring Transfer Mode for mod_plsql Runtime
- Configuring the Runtime Default Locale

About Manually Editing HTTP Server and OC4J Configuration Files

If you edit Oracle HTTP Server or OC4J configuration files manually, instead of using Oracle Enterprise Manager 10*g*, then you must use the DCM command-line utility `dcmctl` to notify the DCM repository of the changes. Otherwise, your changes will not go into effect and will not be reflected in the Enterprise Manager consoles. The commands are as follows:

- To notify the DCM repository of changes made to Oracle HTTP Server configuration files:

  ```
  ORACLE_HOME/dcm/bin/dcmctl updateConfig ohs
  ```

- To notify the DCM repository of changes made to OC4J configuration files:

  ```
  ORACLE_HOME/dcm/bin/dcmctl updateConfig oc4j
  ```

- To notify the DCM repository of changes made to both Oracle HTTP Server and OC4J configuration files:

  ```
  ORACLE_HOME/dcm/bin/dcmctl updateConfig
  ```

Before you change configuration parameters, manually or using Oracle Enterprise Manager 10*g*, you can save the current state of Oracle HTTP Server and OC4J configuration files and installed J2EE applications with the following command:

```
ORACLE_HOME/dcm/bin/dcmctl saveInstance -dir directory_name
```

You can then restore the state and back out of any subsequent changes that were made using the following command:

```
ORACLE_HOME/dcm/bin/dcmctl restoreInstance -dir directory_name
```

Configuring the NLS_LANG Parameter

The `NLS_LANG` parameter controls the language, territory, and character set used for database connections in an Internet application. Specify the value of `NLS_LANG` in the following format, including the punctuation as shown:

```
language_territory.characterset
```

In the preceding syntax, *`language`*, *`territory`*, and *`characterset`* must be valid Oracle language, territory, and character set names. The specified language and territory are used to initialize the locale that determines the default date and time formats, number formats, and sorting sequence in a database session. The Oracle database converts data to and from the specified character set when it is retrieved from or inserted into the database.

> **See Also:** *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library for a list of valid Oracle language, territory, and character set names

You can specify the `NLS_LANG` parameter in the Oracle HTTP Server and OC4J files. The Oracle HTTP Server and OC4J files where `NLS_LANG` can be specified are as follows:

- `$ORACLE_HOME/Apache/Apache/conf/httpd.conf`

  This is the configuration of Oracle HTTP Server powered by Apache, and it defines the environment variables that are passed to Apache modules. If you want

to explicitly specify the NLS_LANG parameter for CGI scripts such as Perl and server-side include (SSI) pages, then you can add the following line to this file:

```
SetEnv NLS_LANG language_territory.characterset
```

Oracle HTTP Server is already configured to use the NLS_LANG shell environment variable in CGI scripts and SSI pages when NLS_LANG is not explicitly specified as described earlier. It does so by putting the following line into the file:

```
PassEnv NLS_LANG
```

- $ORACLE_HOME/Apache/Apache/bin/apachectl

  This is the Oracle HTTP Server startup script used in UNIX. If you want to start Oracle HTTP Server directly from apachectl, then you can specify the following line in this script file to define an NLS_LANG value:

  ```
  NLS_LANG=language_territory.characterset; export NLS_LANG
  ```

- $ORACLE_HOME/opmn/conf/opmn.xml

  Oracle Process Manager and Notification Server (OPMN) is used to manage Oracle HTTP Server and OC4J instances. The opmn.xml configuration file allows you to specify the NLS_LANG environment variable for Oracle HTTP Server and OC4J processes through the following XML construct:

  ```
  <environment>
  ...
  <prop name="NLS_LANG" value="language_territory_characterset" />
  ...
  </environment>
  ```

  This construct can be specified at the Oracle Application Server instance level where it applies to all Oracle HTTP Server and OC4J instances belonging to the Oracle Application Server instance. It can also be specified for the individual Oracle HTTP Server or OC4J instance where it only applies to the corresponding instance.

- $ORACLE_HOME/Apache/modplsql/conf/dads.conf

  This file defines database access descriptors (DADs) for mod_plsql to use when creating a database connection. You can specify the NLS_LANG value for the corresponding DAD. For example, you can specify the NLS_LANG value for the /pls/scott DAD as follows:

  ```
  <Location /pls/scott>
       SetHandler pls_handler
       Order deny, allow
       Allow from all
       PlsqlDatabasePassword          tiger
       PlsqlDatabaseUsername          scott
       PlsqlDocumentPath              docs
       PlsqlNlsLanguage             NLS_LANG value
  </Location>
  ```

  Note that, when the transfer mode of a DAD is CHAR instead of RAW, the NLS_LANG character set of the DAD should be the same as that of the database character set for mod_plsql to work properly.

- $ORACLE_HOME/Apache/Jserv/etc/jserv.properties

  If JServ is needed in your environment, then you need to add or modify the following line in this file to define the appropriate NLS_LANG value:

```
wrapper.env=NLS_LANG=language_territory.characterset
```

If you do not explicitly specify the NLS_LANG environment variable in these files as described in the preceding, Oracle HTTP Server and OC4J will use the value set as follows:

- On UNIX: The NLS_LANG shell environment variable when Oracle HTTP Server and OC4J are invoked
- On Microsoft Windows: The NLS_LANG registry key at \\HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME*n* in the Win32 registry

Preconfigured NLS_LANG Values

The Oracle Application Server installation pre-configures NLS_LANG values in the following Oracle HTTP Server and OC4J files for you based on the locale of the runtime environment on which the product is installed.

- $ORACLE_HOME/Apache/Apache/bin/apachectl (for UNIX platforms)
- $ORACLE_HOME/opmn/conf/opmn.xml/opmnctl (for UNIX platforms)

The pre-configured NLS_LANG values in the apachectl and opmnctl scripts are specified as follows:

```
NLS_LANG=${NLS_LANG=language_territory.characterset}; export NLS_LANG
```

The preceding line means the pre-configured NLS_LANG values are used only when the shell environments from which the scripts are invoked have not defined the NLS_LANG environment variable. If you want to use an NLS_LANG value regardless of the shell environment, then change the line to:

```
NLS_LANG=language_territory.characterset; export NLS_LANG
```

The NLS_LANG parameter controls the locale of the runtime environment on which OPMN runs. It should correspond to the default locale of the middle-tier runtime environment, which is the default locale of the operating system. The same NLS_LANG parameter is inherited by the OPMN managed processes, such as Oracle HTTP Server and OC4J, unless it is explicitly specified with a different value in opmn.xml.

For Microsoft Windows platforms, the pre-configured NLS_LANG is automatically registered in the Win32 registry as the NLS_LANG registry key at \\HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\HOME*n*. The NLS_LANG value in this registry key controls the locale of the runtime environment on which OPMN and its managed processes run.

The pre-configured NLS_LANG values are the best values derived from the runtime locale during product installation, and may not represent the appropriate value for your Oracle HTTP Server and OC4J configurations. You may need to alter these values according to your specific requirements and runtime environments.

Setting NLS_LANG for a Monolingual Application Architecture

Set the NLS_LANG parameter to specify the language, territory, and character set that correspond to the locale that its middle-tier server is configured to serve. If most clients are running on Microsoft Windows platforms, then it is a good practice to use the NLS_LANG character set that corresponds to the Microsoft Windows code page of the locale. For example, when you configure the middle-tier server to serve Japanese clients, then specify the following value for NLS_LANG:

```
JAPANESE_JAPAN.JA16SJIS
```

JA16SJIS corresponds to code page 932 of the Japanese Microsoft Windows operating system.

Table 6–1 lists the NLS_LANG values for the most commonly used locales.

Table 6–1 NLS_LANG Values for Commonly Used Locales

Locale	NLS_LANG Value
Arabic (Egypt)	ARABIC_EGYPT.AR8MSWIN1256
Arabic (U.A.E.)	ARABIC_UNITED ARAB EMIRATES.AR8MSWIN1256
Chinese (Taiwan)	TRADITIONAL CHINESE_TAIWAN.ZHT16MSWIN950
Chinese (P.R.C.)	SIMPLIFIED CHINESE_CHINA.ZHS16GBK
Czech	CZECH_CZECH REPUBLIC.EE8MSWIN1250
Danish	DANISH_DENMARK.WE8MSWIN1252
Dutch	DUTCH_THE NETHERLANDS.WE8MSWIN1252
English (United Kingdom)	ENGLISH_UNITED KINGDOM.WE8MSWIN1252
English (U.S.A.)	AMERICAN_AMERICA.WE8MSWIN1252
Finnish	FINNISH_FINLAND.WE8MSWIN1252
French (Canada)	CANADIAN FRENCH_CANADA.WE8MSWIN1252
French (France)	FRENCH_FRANCE.WE8MSWIN1252
German (Germany	GERMAN_GERMANY.WE8MSWIN1252
Greek	GREEK_GREECE.EL8MSWIN1253
Hebrew	HEBREW_ISRAEL.IW8MSWIN1255
Hungarian	HUNGARIAN_HUNGARY.EE8MSWIN1250
Italian (Italy)	ITALIAN_ITALY.WE8MSWIN1252
Japanese	JAPANESE_JAPAN.JA16SJIS
Korean	KOREAN_KOREA.KO16MSWIN949
Norwegian	NORWEGIAN_NORWAY.WE8MSWIN1252
Polish	POLISH_POLAND.EE8MSWIN1250
Portuguese (Brazil)	BRAZILIAN PORTUGUESE_BRAZIL.WE8MSWIN1252
Portuguese (Portugal)	PORTUGUESE_PORTUGAL.WE8MSWIN1252
Romanian	ROMANIAN_ROMANIA.EE8MSWIN1250
Russian	RUSSIAN_CIS.CL8MSWIN1251
Slovak	SLOVAK_SLOVAKIA.EE8MSWIN1250
Spanish (Spain)	SPANISH_SPAIN.WE8MSWIN1252
Spanish (Latin American)	LATIN AMERICAN SPANISH_AMERICA.WE8MSWIN1252
Swedish	SWEDISH_SWEDEN.WE8MSWIN1252
Thai	THAI_THAILAND.TH8TISASCII
Turkish	TURKISH_TURKEY.TR8MSWIN1254

Setting NLS_LANG for a Multilingual Application Architecture

The language and territory components of the NLS_LANG parameter are not as important in the multilingual application architecture as they are in the monolingual

application architecture. A multilingual application needs to handle different locales dynamically and cannot rely on fixed settings. The application should always use the UTF-8 character set so that Unicode data can be retrieved from and inserted into the database. An example of an appropriate value for NLS_LANG in a multilingual deployment is:

```
NLS_LANG=AMERICAN_AMERICA.UTF8
```

Configuring Transfer Mode for mod_plsql Runtime

The transfer mode of each database access descriptor (DAD) of the mod_plsql runtime enables PL/SQL to construct HTML content and process HTML form input in different character sets. You must set the transfer mode with the appropriate value.

It is important to configure the transfer mode for the mod_plsql module in the $ORACLE_HOME/Apache/modplsql/conf/dads.conf file where the DADs are specified.

The mod_plsql module supports two transfer modes that you can configure in a DAD:

- CHAR mode: This is a default mode where dynamic HTML content is sent as VARCHAR2 data from the database to mod_plsql. In this mode, the NLS_LANG character set must be the same as that of the back-end database character set.
- RAW mode: Dynamic HTML content is sent as RAW data from the database to mod_plsql and is subject to character set conversion in the database server where the PL/SQL procedures and Oracle PL/SQL Server Pages (PSP) run. Character set conversion happens only when the HTML page encoding is specified, either by the NLS_LANG character set or by the charset parameter specified in the OWA_UTIL.MIME_HEADER() function call.

You should turn on the RAW transfer mode in a DAD for both monolingual and multilingual Internet applications as follows:

```
<Location /pls/scott>
        SetHandler pls_handler
        Order deny,allow
        Allow from all
        PlsqlDatabasePassword          tiger
        PlsqlDatabaseUsername          scott
        PlsqlDatabaseConnectString     local
        PlsqlDocumentPath              docs
        PlsqlNlsLanguage               AMERICAN_AMERICA.UTF8
        PlsqlTransferMode              RAW
</Location>
```

If the value of PlsqlNlsLanguage has a space in it, then the value must be enclosed in quotation marks. For example:

```
PlsqlNlsLanguage "SIMPLIFIED CHINESE_CHINA.ZHS16GBK"
```

Configuring the Runtime Default Locale

This section describes how to initialize the runtime default locale for runtime environments that Oracle Application Server supports:

- mod_jserv Runtime for Java
- OC4J Java Runtime

- mod_plsql Runtime for PL/SQL and Oracle PL/SQL Server Pages
- mod_perl Runtime for Perl Scripts
- C/C++ Runtime

The default locale of a runtime environment controls the default locale-sensitive behavior of the applications, such as the character set used in file I/O operations, the language of the user interface, and the date format used. It needs to be properly set in order for applications relying on the default runtime locale to run with the expected locale-sensitive behavior. The default runtime locale is usually inherited from the default locale of the operating system or the locale of the runtime process.

The default runtime locale should be used as the user's preferred locale for monolingual applications. For multilingual applications, the default runtime locale is used for any server-side I/O operations, such as logging messages.

mod_jserv Runtime for Java

For UNIX platforms, the LANG or LC_ALL variable defines the following:

- The POSIX (also known as XPG4) locale used for a process
- How Java VM initializes its default locale

To configure the Java VM for JServ, define the LANG or LC_ALL environment variable with a POSIX locale name in the `jserv.properties` file. For example, the following line in `jserv.properties` defines Japanese (Japan) to be the default locale of Java VM for JServ on UNIX:

```
wrapper.env=LANG=ja_JP
```

The values for the LANG and LC_ALL environment variables should refer to the same POSIX locale available in your operating system. The LC_ALL environment variable always overrides the LANG environment variable if they are different.

The regional settings of the Control Panel control the default locale of the Java VM for JServ on Microsoft Windows platforms. Change the regional settings to the desired locale from the Control Panel before starting Oracle HTTP Server.

OC4J Java Runtime

Define the LANG or LC_ALL environment variable with a POSIX locale name in `$ORACLE_HOME/opmn/conf/opmn.xml`. For example, the following line within the `<environment>` tags in `opmn.xml` defines Japanese (Japan) to be the default locale of Java VM for OC4J on Solaris:

```
<environment>
...
<prop name="LANG" value="ja_JP" />
...
</environment>
```

The regional settings of the Control Panel control the default locale of the Java VM for OC4J on Microsoft Windows platforms. Change the regional settings to the desired locale from the Control Panel before starting Oracle HTTP Server.

mod_plsql Runtime for PL/SQL and Oracle PL/SQL Server Pages

PL/SQL and PSP run on an Oracle database in the context of a database session. Therefore, the NLS_LANG parameter controls the runtime default locale. The NLS_

LANG parameter should be configured as described in "Configuring the NLS_LANG Parameter".

mod_perl Runtime for Perl Scripts

Perl scripts run on the Perl interpreter that the mod_perl module provides. The locale support in Perl is based on the POSIX locale available in the operating system. It uses the underlying POSIX C libraries as a foundation. To configure the Perl runtime default locale, follow the procedure described for the C/C++ runtime.

See Also:

- "C/C++ Runtime"
- *Oracle HTTP Server Administrator's Guide* for more information about how Perl scripts use POSIX locales

C/C++ Runtime

The C/C++ runtime uses the POSIX locale system provided by the operating system. You can configure the locale system by defining the LC_ALL or LANG environment variable. Define LC_ALL with a valid locale value that the operating system provides. These values are different on different operating systems.

See Also: Table 6–1 for a list of commonly used POSIX locales for Solaris

For UNIX platforms, define LC_ALL as follows:

- In the $ORACLE_HOME/Apache/Apache/conf/httpd.conf file, add the following line:

  ```
  PassEnv LC_ALL
  ```

- In the $ORACLE_HOME/Apache/Apache/bin/apachectl file, add the following line:

  ```
  LC_ALL=${LC_ALL=OS_locale}; export LC_ALL
  ```

For Microsoft Windows platforms, the POSIX locale should inherit its value from the regional settings of the Control Panel instead of being specified in the LC_ALL environment variable. Change the regional settings to change the default runtime POSIX locale.

Configuring Oracle Application Server Portal for Global Deployment

Oracle Application Server Portal (OracleAS Portal) is designed to allow application development and deployment in different languages. OracleAS Portal is configured with the languages that are selected in the Oracle Universal Installer during the Oracle Application Server middle-tier installation. The selected languages that are configured show up in the Set Language portlet.

To configure additional languages after installation, use the ptllang script. Once you have installed a language, OracleAS Portal allows you to specify the preferred locale and territory to be used for that language; for example, Australian English or Canadian French.

See Also: *Oracle Application Server Portal Configuration Guide* for information about the ptllang script

Configuring Oracle Application Server Wireless for Global Deployment

When users access wireless services from their mobile devices, Oracle Application Server Wireless uses the user profile information from Oracle Internet Directory to determine the user's preferred language. Administrators can select the language when creating a new user through the Oracle Application Server Wireless Tools. Users can change their preferred language through the Wireless Customization Tool.

Configuring Encoding for Outgoing E-mail Messages

When users send e-mail messages from their mobile devices, Oracle Application Server Wireless sends the messages in the encoding specified in the encoding parameter of the PIM / Mail service.

You can change the default encoding for outgoing e-mail messages by modifying the ORACLE_SERVICES_PIM_MAIL_MESSAGE_ENCODING parameter of the PIM / Mail master service.

Configuring Oracle Application Server Forms Services for Global Deployment

The NLS_LANG parameter controls the language, territory, and character set that an Internet application uses for database connections. Specify the value of NLS_LANG in the following format, including the punctuation as shown:

```
language_territory.characterset
```

language, *territory*, and *characterset* must be valid Oracle language, territory, and character set names. The specified language and territory are used to initialize the locale that determines the default date and time formats, number formats, and sorting sequence in a database session. Oracle Net converts data to and from the specified character set when it retrieves data from or inserts data into the database.

You can set the NLS_LANG parameter in the $ORACLE_HOME/forms/server/default.env file. If you do not set the NLS_LANG parameter in the default.env file, then OracleAS Forms Services uses the value set as follows:

- On UNIX: The NLS_LANG shell environment variable when OracleAS Forms Services is invoked
- On Microsoft Windows: The NLS_LANG setting at the \\HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE*FormsServerOracle_HOME* in the Microsoft Windows registry

You can have different NLS_LANG settings on the same Forms Server by specifying an alternate environment file. Use the envFile parameter in the formsweb.cfg file. To do this:

1. Create two environment configuration files under $ORACLE_HOME/forms/server. For example, an American environment configuration file (en-us.env) should contain the following lines:

   ```
   NLS_LANG=AMERICAN_AMERICA.US7ASCII
   FORMS_PATH=d:\us
   ```

   A Japanese environment configuration file (ja.env) should contain the following lines:

   ```
   NLS_LANG=JAPANESE_JAPAN.JA16SJIS
   FORMS_PATH=d:\ja
   ```

2. In the $ORACLE_HOME/forms/server/formsweb.cfg file, set the envFile parameter for the alternative setting. For example:

```
[ja]
envFile=ja.env

[en-us]
envFile=en-us.env
```

3. Specify the configuration name in the URL for your forms servlet as follows:

```
http://formsservermachine/forms/frmservlet?config=ja
http://formsservermachine/forms/frmservlet?config=en
```

See Also: *Oracle Application Server Forms Services Deployment Guide*

Configuring OracleAS Reports Services for Global Deployment

The NLS_LANG parameter controls the language, territory, and character set used for database connections in an OracleAS Reports Services application. Specify the value of NLS_LANG in the following format, including the punctuation as shown:

```
language_territory.characterset
```

language, *territory*, and *characterset* must be valid Oracle language, territory, and character set names. The specified language and territory are used to initialize the locale that determines the default date and time formats, number formats, and sorting sequence in a database session. Oracle Net converts data to and from the specified character set when it retrieves data from or inserts data into the database.

OracleAS Reports Services uses the value of the NLS_LANG parameter set as follows:

- On UNIX: The NLS_LANG shell environment variable when OracleAS Reports Services is invoked. The default NLS_LANG value is set in $*ORACLE_HOME*/bin/reports.sh.
- On Microsoft Windows: The value of NLS_LANG set at \\HKEY_LOCAL_MACHINE\SOFTWARE\ORACLE\%*ReportsORACLE_HOME%* in the Microsoft Windows registry
- For dynamic environment switching: In the OracleAS Reports Services configuration file through the environment element. On UNIX, the default NLS_LANG value in the reports.sh file needs to be commented out to enable this feature.

See Also:

- *Oracle Application Server Reports Services Publishing Reports to the Web* for more information about dynamic environment switching
- *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library for more information about these parameters

Configuring Oracle Business Intelligence Discoverer for Global Deployment

OracleBI Discoverer can simultaneously support users with different locales. Users may explicitly control the locale used for the user interface, or they may allow

OracleBI Discoverer to automatically determine a default. The order of precedence for determining the language and locale is:

1. Language and locale settings specified in the URL for OracleBI Discoverer.
2. Language and locale settings specified in the OracleBI Discoverer Connection. If the Locale set in user's browser option is specified, then the language settings in the each user's browser is used.
3. Language and locale of Oracle Application Server.

> **See Also:** *Oracle Business Intelligence Discoverer Configuration Guide* for more information on using URL parameters with OracleBI Discoverer.

Configuring a Centralized Unicode-enabled Database to Support Global Deployment

You can set up the centralized Oracle database to store Unicode data in the following ways:

- As UTF-8 in the SQL CHAR data types (CHAR, VARCHAR2, and CLOB)
- As UTF-16 in the SQL NCHAR data types (NCHAR, NVARCHAR2, and NCLOB)

It is good practice to specify the centralized Oracle database to support the following data types:

- Specify AL32UTF8 for the database character set when you create the centralized database for UTF-8 in the SQL CHAR data types
- Specify AL16UTF16 for the national character set when you create the centralized database for UTF-16 in the SQL NCHAR data types

Example 6–1 shows part of a CREATE DATABASE statement that sets the recommended database character set and national character set.

Example 6–1 Specifying the Database Character Set and the National Character Set

```
CREATE DATABASE utfdb
    CONTROL FILE REUSE
    LOGFILE '/u01/oracle/utfdb/redo01.log' SIZE 1M REUSE
    '/u01/oracle/utfdb/redo02.log' SIZE 1M REUSE
    DATAFILE '/u01/oracle/utfdb/system01.dbf' SIZE 10M REUSE
    AUTOEXTENT ON
    NEXT 10M MAXSIZE 200M
    CHARACTER SET AL32UTF8
    NATIONAL CHARACTER SET AL16UTF16
    ... ;
```

> **See Also:**
>
> - *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library
> - *Oracle Database SQL Reference* in the Oracle Database Documentation Library

7

A Multilingual Demonstration for Oracle Application Server

This chapter describes the World-of-Books demonstration that is provided with Oracle Application Server. The demonstration utilizes the Oracle Globalization Kit (GDK).

This chapter contains the following topics:

Description of the World-of-Books Demonstration

The World-of-Books (WOB) demonstration illustrates how to write a multilingual Web application and deploy it on the Oracle Application Server J2EE container. The application consists of the following Web sites:

- An online store that sells books in different languages
- An online Chinese book supplier administration site that represents book Supplier A
- An online global book supplier administration site that represents book Supplier B

The online bookstore is a multilingual Web application that interacts with customers. It allows customers to view books, check prices, and place orders. The application uses HTTP connections to send orders as XML documents to the suppliers. The online book supplier administration sites are Web applications that the book suppliers use to get orders from the bookstore, to send order status reports to the bookstore, and to notify the bookstore about newly available books.

The online bookstore supports 60 locales. Customers in these locales can use the online bookstore with their preferred language and cultural conventions. The online book supplier administration sites are in English only.

Architecture and Design of the World-of-Books Demonstration

The WOB demonstration serves customers with different locale preferences. It is mainly written in Java, using Java Servlets, Java beans, and Java Server Pages (JSPs). It uses the Unicode capabilities available in Oracle GDK, XML, JDBC, and the Oracle database to support multilingual data and a multilingual user interface.

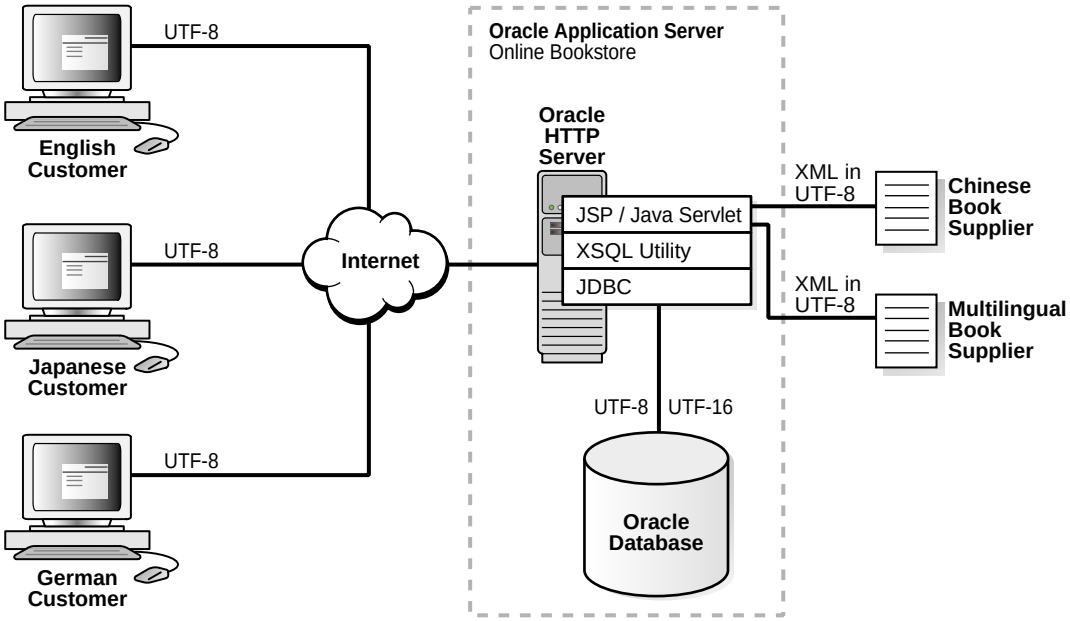
This section contains the following topics:

- World-of-Books Architecture
- World-of-Books Design
- World-of-Books Schema Design

World-of-Books Architecture

Figure 7–1 shows the architecture of the WOB demonstration.

Figure 7–1 World-of-Books Architecture



The application architecture can be summarized as follows:

- JSPs generate dynamic content in UTF-8 encoded HTML pages.
- Java Servlets and Java Beans implement the business logic.
- The Oracle database stores book and customer information.
  - Oracle Text enables locale-sensitive, full-text searches on the contents of books.
  - The SQL NVARCHAR2 datatype stores multilingual book information.
- The Oracle JDBC driver (either OCI or Thin driver) accesses Unicode data stored in the Oracle database. The data can be encoded in UTF-8 if the target column is of

SQL CHAR datatype, or the data can be encoded in UTF-16 if the target column is of SQL NCHAR datatype.

- The document format for communications between the online bookstore and the book suppliers is UTF-8 encoded XML.

Figure 7–1 shows the WOB application on Oracle Application Server. The processing character set for the WOB application is UTF-16. The application uses XML messages to communicate with the Chinese book supplier and the multilingual book supplier. The XML messages are encoded in the UTF-8 character set. English, Japanese, and German customers connect to the WOB application through the Internet. The application serves all customers HTML pages encoded in the UTF-8 character set.

World-of-Books Design

Table 7–1 shows the Java programs that contain most of the globalization features for the WOB application. The programs are located in the $WOB_HOME/src/oracle/demo/wob2/wob directory.

Table 7–1 Java Programs that Contain Globalization Features for the World-of-Books Application

Java Program	Purpose
beans/LocalizationContext.java	Contains locale-sensitive methods for a specific user session
ApplicationLocales.java	Provides convenient methods based on the application locales configured in the Oracle GDK configuration file

The LocalizationContext bean uses the Oracle GDK to expose locale-sensitive behavior to the WOB application. The Oracle GDK provides a framework to build enabled multilingual Web applications. The Java ApplicationLocales class manages a set of locales for an application and provides services based on these locales. This class provides convenience methods based on the application locales configured in the Oracle GDK configuration file gdkapp.xml.

Most of the JSPs for the online bookstore include the header.jsp file, which uses the LocalizationContext Java bean to keep locale information for a session. JSPs call the LocalizationContext Java bean to perform all locale-sensitive operations such as formatting a date, encoding a URL, and converting HTML form parameters to Java strings.

World-of-Books Schema Design

The database schema for the WOB demonstration consists of the following tables:

- customers: Stores the user profile for each WOB user.
- books: Stores the information about each book.
- docs: Stores the content of each book so that customers can search the content of the books.

Table 7–2 describes the customers table. When a registered user is logged in, the online bookstore uses the locale preferences in the customer table in the LocalizationContext bean.

Table 7–2 Description of the customers Table

Column	Datatype	Description
currency1	VARCHAR2(10)	ISO locale whose default primary currency is used by the user
currency2	VARCHAR2(10)	ISO locale whose default dual currency is used by the user
custid	VARCHAR2(50)	User's name (this is the primary key)
locale	VARCHAR2(10)	User's preferred locale, in ISO locale format (for example, en-US)
timezone	VARCHAR2(50)	User's time zone (for example, Asia/Hong Kong)

Table 7–3 describes the books table. The NVARCHAR2 datatype is used for the title, author, short description, and publisher of the book. By storing this information as Unicode in the NVARCHAR2 datatype, the WOB demonstration can support books in languages from around the world. The nsort column is used for queries about books so that the list is returned in an order appropriate for the locale.

Table 7–3 Description of the books Table

Column	Datatype	Description
author	NVARCHAR(300)	Book author
bookid	NUMBER (10)	Unique identifier of the book (this is the primary key)
descpt	NVARCHAR(2000)	Short description of the book
langid	NUMBER(3)	Language of the book
nsort	VARCHAR2(30)	Locale-sensitive sorting sequence used in the NLSSORT() SQL function for the book
publisher	NVARCHAR(200)	Name of the book publisher
title	NVARCHAR(300)	Book title

Table 7–4 describes the docs table. It stores the contents of the books.

Table 7–4 Description of the docs Table

Column	Datatype	Description
bookid	NUMBER(10)	Unique identifier of the book (this is the primary key)
cset	VARCHAR2(30)	Character set of the contents of the book
doc	BLOB	Contents of the book
format	VARCHAR2(10)	Format of the contents of the book (TEXT or BINARY)
langid	NUMBER(3)	Language of the book
language	VARCHAR2(30)	Language of the contents of the book, using the Oracle Globalization Support language naming convention
mimetype	VARCHAR2(50)	MIME type of the book

Indexes have been built for these tables. The following SQL files are used to create these tables and build the corresponding indexes. They are located in the $WOB_HOME/schema directory:

- customers.sql

- books.sql
- docs.sql

Oracle Text requires the language, format, cset, and doc columns of the docs table to build a full-text search index on the docs table. The ctxidx.sql and ctxsys.sql scripts are used to set up the full-text search index. They are located in $WOB_HOME/schema/ctx.

> **See Also:** *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library for more information about building a full-text search index

Installing the World-of-Books Demonstration

The World-of-Books (WOB) demonstration is available as a zip file you can download from the Oracle Web site at

http://www.oracle.com/technology/tech/java/oc4j/demos/index.html

After you download the globalization_wob_demo.zip file, unzip the file as follows:

1. Go to the $ORACLE_HOME/j2ee/home/demo directory, or create it if it does not already exist.
2. Copy the file to the $ORACLE_HOME/j2ee/home/demo directory.
3. Unzip the file.

> **Note:** Environment variable references, such as $ORACLE_HOME, are shown in UNIX format. For Microsoft Windows environments, use the %ORACLE_HOME% notation.

After unzipping the downloaded file, you should see the directory globalization under $ORACLE_HOME/j2ee/home/demo. The directory, $ORACLE_HOME/j2ee/home/demo/globalization, is the root directory of the WOB demonstration. This root directory is referred to as $WOB_HOME throughout this chapter.

Building, Deploying, and Running the World-of-Books Demonstration

The source code and the build files of the WOB demonstration are in the WOB demo home directory located in $WOB_HOME. Table 7–5 shows the directory structure under $WOB_HOME.

> **Note:** Environment variable references, such as $ORACLE_HOME, are shown in UNIX format. For Microsoft Windows environments, use the %ORACLE_HOME% notation.

Table 7–5 World-of-Books Directory Structure

Directory/Files	Description
build.xml	Builds the WOB demonstration

Table 7–5 (Cont.) World-of-Books Directory Structure

Directory/Files	Description
docroot	Contains all static files such as HTML files, JSPs, and images
docroot/suppa	Contains static files for the Chinese book supplier administration application
docroot/suppb	Contains static files for the global book supplier administration application
docroot/wob	Contains static files for the online bookstore Web application
etc	Contains the configuration files for the WOB demonstration applications
j2ee_config	Contains J2EE deployment files for the WOB demonstration
README.TXT	Contains useful information for building and deploying the WOB demonstration
schema	Contains SQL files to create and populate the database schema that the WOB demonstration uses
src/oracle/demo/wob2	Contains all Java programs
src/oracle/demo/wob2/supp	Contains Java programs shared by the two online supplier applications
src/oracle/demo/wob2/wob	Contains Java programs for the online bookstore application

This section contains the following topics:

- How to Build the World-of-Books Demonstration
- How to Deploy the World-of-Books Demonstration
- How to Run the World-of-Books Demonstration

How to Build the World-of-Books Demonstration

To build the WOB demonstration:

1. Go to the `$ORACLE_HOME/j2ee/home/demo/globalization` directory.
2. Update the `suppa.properties`, `suppb.properties`, and `wob.properties` files in the `$WOB_HOME/etc` directory.
   - Replace `<J2EE_HOME>` with the full path where OC4J is installed.
   - Replace `<HOSTNAME>` with the host name of your machine.
   - Replace `<PORT>` with the port number of your default Web site.
3. Set up the JAVA build environment by defining the `JAVA_HOME` and `CLASSPATH` environment variables. Oracle Application Server bundles JDK under `$ORACLE_HOME/jdk` so that you can use it for your `JAVA_HOME`.

   You can also use your own JDK. For example:

```
% setenv ORACLE_HOME yourOracleHome
% setenv JAVA_HOME $ORACLE_HOME/jdk
% setenv J2EE_HOME $ORACLE_HOME/j2ee
% setenv J2EE_HOME $ORACLE_HOME/j2ee/home
```

```
% mkdir $J2EE_HOME/applib
% copy $ORACLE_HOME/lib/xsu12.jar to $J2EE_HOME/applib
% copy $ORACLE_HOME/rdbms/jlib/xdb.jar to $J2EE_HOME/applib
% copy $ORACLE_HOME/jlib/regexp.jar to $J2EE_HOME/applib
```

4. Make sure that `$ORACLE_HOME/bin` and `$ORACLE_HOME/jdk/bin` are in your directory path. For example:

   ```
   % setenv PATH $ORACLE_HOME/bin:$ORACLE_HOME/jdk/bin:$PATH
   ```

5. Ensure that an Oracle10*g* database is available to load the schema and data for the WOB demonstration by defining the `TWO_TASK` environment variable to point to the database. For example, if you can access the database from SQL*Plus with the connect string `iasdb`, then you can define appropriate environment variable to point to the connect string.

   - For UNIX, set the `TWO_TASK` environment variable as follows:

     ```
     % setenv TWO_TASK iasdb
     ```

   - For Microsoft Windows, set the `LOCAL` environment variable as follows:

     ```
     set LOCAL=iasdb
     ```

6. Build the demonstration schema by entering the `ANT` command from the `$WOB_HOME` directory.

   ```
   % ant setupschema
   ```

7. Build the demonstration EAR and WAR files by entering the following `ant` command under the `$J2EE_HOME/demo/globalization` directory:

   ```
   % ant
   ```

   The build process performs the following tasks:

   - Compiles all Java programs
   - Creates the WOB schema and populates it with the seed data that is provided
   - Packages all static files and Java classes into an EAR file and a WAR file, which are used for deployment

8. If you enabled Oracle Text in your database, then you can set up full text searches on book content by building the full text search index using the following command.

   ```
   % ant setupctx
   ```

How to Deploy the World-of-Books Demonstration

To deploy the WOB demonstration on Oracle Application Server J2EE:

1. Update `$WOB_HOME/j2ee_config/data-sources.xml`, which is used for database connection.

   - Replace `HOSTNAME` with the host name of the Oracle database server.
   - Replace `PORT` with the port number of the Oracle database server.
   - Replace `ORACLE_SID` with the system identifier of the Oracle database server.
   - Cut and paste the contents of the `data-sources.xml` file into the `$J2EE_HOME/config/data-sources.xml` file.

2. Update the configuration file using DCM as follows:

```
dcmctl updateConfig
```

3. Deploy the application `$WOB_HOME/lib/glln.ear` using Oracle Enterprise Manager. Alternatively, you can deploy the application using `dcmctl` as follows:

```
dcmctl deployApplication -file $WOB_HOME/lib/glln.ear
                         -application glln -component home
```

How to Run the World-of-Books Demonstration

The online bookstore requires one of the following browsers:

- Internet Explorer 5.5 or later
- Mozilla 1.5 or later
- Netscape 7.0.1 or later

The book supplier administration applications require Internet Explorer 5.5 or later.

To run the WOB demonstration, start the browser and enter the following URL:

`http://`*`host_name`*`:7778/g11n/imap.html`

You should see a screen similar to the following:

Select a link to start the desired application.

Image	Link Target
World-of-Books image	Online bookstore application
Supplier A image	Chinese book supplier administration application
Supplier B image	Global book supplier administration application

You can navigate the online bookstore as a registered customer or as a visitor.

If you click the Supplier B image, the following screen appears:

The links on the Supplier B administration site are as follows:

Link	Description
Update Catalog	Allows the supplier to send new book information to the online bookstore to update the bookstore catalog. It sends an XML file to the online bookstore.
Order Table	Allows the supplier to check for customer orders sent from the online bookstore and can update the order status.
Clean up	Restores the data to the initial state. All previous orders and newly added books are deleted.
XML dir	Lists the XML documents that have been sent to and from the online bookstore.
Home	Returns to the WOB home page.

Locale Awareness of the World-of-Books Demonstration

The World-of-Books online bookstore is fully aware of the user's locale. The application determines the user's locale and uses this locale to format dynamic HTML pages according to the user's language and cultural conventions.

This section contains the following topics:

- How World-of-Books Determines the User's Locale
- How World-of-Books Uses Locale Information in LocalizationContext Methods
- How World-of-Books Sorts Query Results
- How World-of-Books Searches the Contents of Books

How World-of-Books Determines the User's Locale

The online bookstore determines the user's locale using three methods in the following order:

1. If a customer has logged into the bookstore, it examines the locale associated with the customer's user profile and uses it as the preferred locale.

2. Allows the user to enter the locale from the bookstore's user interface.
3. Examines the Accept-Language HTTP header sent from the browser.

The World-of-Books application determines the client's user interface language using the GDK application framework support for locale determination capabilities. This support is provided by the use of predefined `<locale-sources>` XML tag declarations in the GDK configuration file `gdkapp.xml`. The file resides in the `$WOB_HOME/docroot/WEB_INF` directory.

The `<locale-determine-rule>` XML tag in the `gdkapp.xml` file allows the developer to configure various sources of the locale for the Web application user interface language. The WOB `gdkapp.xml` configuration file syntax for configuring locale source looks like the following:

```
<locale-determine-rule>
  <db-locale-source data source-name="jdbc/WobWobDataSource"
     locale-source-table="customers"
     user-key="userinfo" user-column="custid" locale-column="locale"
     timezone-column="timezone">
     oracle.i18n.servlet.localesource.DBLocaleSource
  </db-locale-source>
  <locale-source>oracle.i18n.servlet.localesource.UserInput</locale-source>
  <locale-source>oraclei18n.servlet.localesource.HttpAcceptLanguage
  </locale-source>
</locale-determine-rule>
```

The locale determination model for the WOB is evaluated as follows:

- If the user is logged in, then the `<db-locale-source>` XML element attribute `locale-column` determines the user interface language, such as locale. In the WOB case, the `locale` column in the customer table determines the user's locale preference. This scenario is run once the user is authenticated and logged into the WOB application.
- If the user is not logged in, then the user interface language is determined by any locale specified by the user through a menu or as a parameter in the HTTP protocol, such as an HTTP query string parameter. This is the case when a user clicks a flag for a particular country on the WOB welcome page. If the user is not logged in and does not explicitly input locale information, then the user interface language is determined by the HTTP protocol `Accept-Language` header value.
- The `LocalizationContext` bean composes the GDK functionality to provide services that help with localization of the user interface for a multilingual user. The `LocalizationContext` bean contains a reference to the GDK `Localizer` object for the current HTTP request, and uses it for retrieving localized messages and getting access to the predetermined locale. It uses the other GDK API for formatting dates, currencies, displaying timezones, getting other localized display names and retrieving other locale-specific attributes.

How World-of-Books Uses Locale Information in LocalizationContext Methods

After the `Localizer` is initialized with the user's locale, all methods of the `LocalizationContext` are sensitive to the locale. Table 7–6 shows examples of locale-sensitive methods defined in the `LocalizationContext`.

Table 7–6 Examples of Locale-Sensitive Methods of the Localizer Bean

Method	Example of Use
`String formatDate()`	The following JSPs use the `formatDate()` method: ■ `welcome.jsp` formats the system date that the welcome page displays. ■ `History.jsp` formats the date of the order history. ■ `setting.jsp` formats the date to be displayed when a registered user updates the user profile.
`String getCurrency()`	`Changeprofile1.jsp` gets the primary currency symbol to be displayed for the user profile modification screen.
`String getDirection()`	`setting.jsp` displays the direction that text is written, based on the current user.
`String getDualCurrency()`	`Changeprofile1.jsp` gets the alternate or dual currency symbol to be displayed for the user profile modification screen.
`String getMessage()`	Most of the JSPs use this method to get the translated message that corresponds to the current locale from a resource bundle.
`String getNLSLanguage()`	`search.jsp` gets the Oracle language name used for the current locale and for submitting a language-sensitive search.
`String getTimeZone()`	`myaccount.jsp` displays the time zone of the current user.

Other locale-sensitive functions are described in the following sections.

How World-of-Books Sorts Query Results

The order in which books are listed in the results of a query is sensitive to the current user's locale. The search template is as follows:

```
SELECT books.bookid,
       langmap.language,
       books.title,
       books.author,
         substr(books.descpt, 1, 50)
FROM  books, langmap
WHERE  specific search criteria
           books.langid = langmap.langid AND
           nlssort(books.title, 'NLS_SORT = '|| books.nsort) IS NOT NULL
ORDER BY  langord(books.langid, 'Oracle_NLS_language'),
           nlssort(books.title, 'NLS_SORT='||books.nsort);
```

The `langmap` table maps language IDs to Oracle Globalization Support language names and Oracle sort names used in the `NLSSORT` SQL function. The `$WOB_HOME/schema/langmap.sql` file creates the `langmap` table.

The `SELECT` statement orders the books with the `ORDER BY` clause as follows:

1. It groups the books by their languages, using the first sort key that the `langord` PL/SQL function returns. The `langord` function returns the smallest key value when the Oracle Globalization Support language that corresponds to the current user's locale matches the language of the book. Thus the books are grouped so that the first group consists of books whose language corresponds to the user's locale.
2. Within each language group, it orders the books by the sort key that the `NLSSORT` SQL function returns. The `NLSSORT` function generates sort keys based on the linguistic order specified by the `NLS_SORT` parameter. The value of the `NLS_SORT`

parameter is stored in the `nsort` column of the `books` table. Thus the books in the sorted group are ordered by the Oracle sort sequence name stored in the `nsort` column.

The application also orders lists in the user interface using locale information. For example, it uses the `displayLanguageOptions()` method of the `LocalizationContext` bean to construct a list of languages so users can select a language. The `displayLanguageOptions()` method collates the languages in the list based on the current locale as determined by the GDK Localizer. A sample of the `displayLanguageOptions()` code is as follows:

```
public String displayLanugageOptions()
{
    Set set = appLocales.getSupportedLanguagesByStyle(locale,
    ApplicationLocales.LANGUAGE_DISPLAY_ITEM):
    StringBuffer optionBuffer = new StringBuffer();
    Iterator iter = set.iterator();
    OraDisplayLocaleInfo odli = OraDisplayLocaleInfo.getInstance(locale);

    while (iter.hasNext())
    {
       OraLocaleInfo oli = (OraLocaleInfo) iter.next();
       String oliLang = oligetLocale().getLanguage();
       option.Buffer.append("<option value=\"" + oliLang + "\"" +
       (locale.getLaguage().equals(oliLang) ? " selected" : "")
       + ">" + odli.getDisplayLangauge(oli.getDisplayLanguage(oli.getLocale()) +
       " [" + oliLang + "]</option>\n");
    }
    return optionBuffer.toString();
}
```

The `ApplicationLocales` Java class provides a different locale view subset on the application locales that are declared in the `gdkapp.xml` configuration file. In the preceding code sample, the `getSupportedLanguagesByStyle()` method returns a set of `OraLocaleInfo` objects representing the supported languages of the WOB. Additionally, the items in the set are sorted by the language display name using the `GDK OraCollator` class. The collated `OraLocaleInfo` objects are then used to generate HTML code for the language select list.

The other methods that collate lists are `displayCountryOptions()`, `displayCurrencyOptions()`, and `displayScriptCountryVars()`.

How World-of-Books Searches the Contents of Books

The online bookstore allows users to search the contents of books in a locale-sensitive manner. The following query searches the contents of the books from the `docs` table:

```
   SELECT  books.bookid,
           langmap.language,
           books.title,
           books.author,
           substr(books.descpt, 1, 50)
     FROM  books, langmap, docs
    WHERE  contains(docs.doc, 'search_key', 0) > 0 AND
           books.langid = langmap.langid AND
           nlssort(books.title, 'NLS_SORT = '|| books.nsort) IS NOT NULL

 ORDER BY  langord(books.langid, 'Oracle_NLS_language'),
           nlssort(books.title, 'NLS_SORT='||books.nsort);
```

The query `(docs.doc,'search_key', 0)` function in the `WHERE` clause returns a positive value when the search key is found in the contents of a document stored in the `doc` column of the `docs` table. The rest of the query is similar to the query used for the book search.

Oracle Text by default uses the language of the search key as defined by the `NLS_LANGUAGE` session parameter. To conduct the search in a language-sensitive manner, `search.jsp` issues an `ALTER SESSION` statement to change the value of the `NLS_LANGUAGE` parameter to the value that the user specifies before submitting the content search query. The `ALTER SESSION` statement is as follows:

```
ALTER SESSION SET NLS_LANGUAGE=language;
```

Calling the `getParameter("v_language")` method of the `HTTPServletRequest` object obtains the language value, where `v_language` is a form input parameter from the advanced search screen.

Encoding HTML Pages for the World-of-Books Demonstration

In the online bookstore, an attribute of the `LocalizationContext` bean stores the encoding used for HTML pages. The `<page-charset>` XML tag in the GDK configuration file is used to specify supported encoding. Currently only one (UTF-8) is defined and set as the default. By default, the online bookstore uses UTF-8 as the HTML page encoding to provide support for multilingual content.

```
<page-charset default="yes">UTF-8</page-charset>
```

By declaring the preceding XML code, the Oracle GDK automatically sets the character set for the HTTP request and response objects to this value at runtime, which in this case is UTF-8.

Handling HTML Form Input for the World-of-Books Demonstration

The online bookstore accepts multilingual text as HTML form input. The input can be a search key when the user wants to search for a book, or it can be a user name at login. The browser sends form input as a sequence of bytes in the same encoding as the HTML form. Page encoding is required to convert the input fro Unicode-encoded Java strings. The page encoding is automatically recognized and converted by the GDK classes at runtime so the correct conversion occurs for the input.

Formatting HTML Pages in the World-of-Books Demonstration

The online bookstore uses the following locale-sensitive text formatting elements for HTML pages:

- Font family
- Writing direction
- Text alignment

To support multiple locales simultaneously, the online bookstore externalizes these elements to locale-specific cascading style sheet (CSS) files instead of hard-coding them in the JSPs. The CSS file structure is the same as the static HTML file structure for the WOB online help.

The CSS files are as follows:

- `$WOB_HOME/docroot/wob/css/style.css` (the default CSS)

- $WOB_HOME/docroot/wob/css/ar/style.css
- $WOB_HOME/docroot/wob/css/he/style.css
- $WOB_HOME/docroot/wob/css/iw/style.css
- $WOB_HOME/docroot/wob/css/ja/style.css
- $WOB_HOME/docroot/wob/css/zh/style.css

In $WOB_HOME/docroot/wob/jsp/header.jsp, the getLocalizedURL() method of the LocalizationContext bean gets the full path of the CSS that corresponds to the current locale. If there is no CSS that is specific to the locale, then the application uses the default CSS.

The following is a sample from the CSS for Arabic text:

```
html { direction: rtl }
h3 { font-size: 100%;
     text-align: end;
     font-weight: bold;
     color: #FFFFFF }
```

The Arabic CSS defines the writing direction of the HTML page as right to left (RTL). The text is always aligned to the end of the writing direction.

The following is a sample from the CSS for Japanese text:

```
html { direction: ltr }
h3 { font-size: 100%;
     text-align: end;
     font-family: "MS Gothic", "MS Mincho", "Times New Roman"…
     font-weight: bold;
     color: #FFFFFF }
tr { font-family: "MS Gothic", "MS Mincho", "Times New Roman",…
     font-size: 12pt; }
p {  font-family: "MS Gothic", "MS Mincho" "Times New Roman",…
     font-size: 12pt}
```

The Japanese CSS defines the writing direction as left to right (LTR). The text is aligned to the end of the writing direction. The font families that are used for displaying Japanese text are MS Gothic and MS Mincho. These are Japanese Microsoft Windows fonts. If you do not specify the font family in the CSS, then the application uses the default font of the browser.

Encoding URLs in the World-of-Books Demonstration

All URLs that are embedded in an HTML page must be encoded. They must use the same encoding as the HTML page. The LocalizationContext bean is the best place to encapsulate the encodeURL() method. This method encodes a URL according to the encoding attribute of the LocalizationContext bean.

The following JSPs call the encodeURL() method:

- Item.jsp
- OrderItem.jsp
- Search.jsp

All embedded URLs for the online bookstore are encoded in ASCII and do not need additional encoding. The encodeURL() method is called to illustrate the concept of encoding URLs.

Accessing the Database in the World-of-Books Demonstration

The WOB demonstration uses the Oracle JDBC driver to access an Oracle database. The JDBC driver transparently converts the data stored in the database to and from Java strings. No special handling is necessary to access Unicode data stored in the database in most cases.

Note: Special handling is required for a Java string bound to a column of the `NVARCHAR` datatype in an `INSERT` or `UPDATE` SQL statement. Use the `setFormOfUse()` method of the `OraclePreparedStatement` class to indicate to JDBC that the target column is of the `NVARCHAR` datatype.

The `setFormOfUse()` method is called in `$WOB_HOME/src/oracle/demo/wob2/supp/beans/insertItem.java` when a new book is inserted into the `books` table.

Organizing the Content of HTML Pages in the World-of-Books Demonstration

The online bookstore consists of the following translatable content:

- Online help as static HTML and image files
- Strings or messages stored for use in composing an HTML page
- Dynamic book information such as the book name and author

This section contains the following topics:

Static Files for World-of-Books Online Help

The static HTML files for the WOB online help are located in `$WOB_HOME/docroot/wob/help`. The English version of the online help is stored at the top level of the `help` directory. The translated help for each locale is stored in the corresponding `help/`*`locale_name`* directory. For example, the Japanese online help is stored in the `help/ja_JP` directory.

The current user's locale determines which help subdirectory the application uses. The `LocalizationContext` bean stores the user's current locale. The `getLocalizedURL()` method returns the correct path of an HTML file that corresponds to the user's locale. For example, given the relative help path of `../help/index.html` and the current locale of `ja_JP`, this method checks for existence of the following files in the order they are listed and returns the first one it finds:

- `$WOB_HOME/docroot/wob/help/ja_JP/index.html`
- `$WOB_HOME/docroot/wob/help/ja/index.html`
- `$WOB_HOME/docroot/wob/help/index.html`

The `header.jsp` file calls this method to get the correct path for every translated HTML file and uses the result to construct the `HREF` tag to reference the appropriate online help.

Using Resource Bundles for the Content of World-of-Books HTML Pages

A list resource bundle stores all translatable messages that comprise the online bookstore user interface. The resource bundle is located in `$WOB_HOME/src/oracle/demo/wob2/wob /resource/MessageBundle.java`. This resource bundle is translated into 27 languages, and the translated resource bundle names have suffixes that correspond to the Java locale name.

The `getMessage()` method of the `LocalizationContext` bean gets a translated message from the resource bundle that corresponds to the current locale. Most JSPs call this method. This method is really a wrapper which ends up calling the GDK Localizer class to actually retrieve the message.

The GDK provides a way for an application to configure the resource bundles it may need. One of the resource bundles can be set to be a default resource bundle for the application. The WOB defines its default resource bundle as follows:

```
oracle.demo.wob2.wob.resource.MessageBundle

<message-bundles>
  <resource-bundle name =
  "default">oracle.demo.wob2.wob.resource.MessageBundle</resource-bundle>
</message-bundles?
```

After the configuration in the preceding example, calling the `getMessage()` method of the `LocalizationContext` bean will retrieve all messages from the default resource bundle `oracle.demo.wob2.wob.resource.MessageBundle`.

A

Oracle Application Server Translated Languages

The following languages are translated for use with Oracle Application Server. Oracle Application Server provides runtime support for more languages than those into which Oracle Application Server itself is translated. For a list of all supported languages, see the *Oracle Database Globalization Support Guide 10g Release 1 (10.1)* in the Oracle Database Documentation Library.

Table A–1 Translated Languages and Abbreviations

Language	Oracle Language Abbreviation
ARABIC	ar
BRAZILIAN PORTUGUESE	ptb
CANADIAN FRENCH	frc
CZECH	cs
DANISH	dk
DUTCH	nl
FINNISH	sf
FRENCH	f
GERMAN	d
GREEK	el
HEBREW	iw
HUNGARIAN	hu
ITALIAN	i
JAPANESE	ja
KOREAN	ko
LATIN AMERICAN SPANISH	esa
NORWEGIAN	n
POLISH	pl
PORTUGUESE	pt
ROMANIAN	ro
RUSSIAN	ru
SIMPLIFIED CHINESE	zhs

Table A–1 (Cont.) Translated Languages and Abbreviations

Language	Oracle Language Abbreviation
SLOVAK	sk
SPANISH	e
SWEDISH	s
THAI	th
TRADITIONAL CHINESE	zht
TURKISH	tr

B

GDK Error Messages

The appendix lists the error messages for the Globalization Developer Kit (GDK).

GDK-03001 Invalid or unsupported sorting rule

Cause: An invalid or unsupported sorting rule name was specified.

Action: Choose a valid sorting rule name and check the globalization guide for the list of sorting rule names.

GDK-03002 The functional-driven sort is not supported.

Cause: A functional-driven sorting rule name was specified.

Action: Choose a valid sorting rule name and check the globalization guide for the list of sorting rule names.

GDK-03003 The linguistic data file is missing.

Cause: A valid sorting rule was specified, but the associated data file was not found.

Action: Make sure the GDK jar files are correctly installed in the Java application.

GDK-03005 Binary sort is not available for the specified character set .

Cause: Binary sorting for the specified character set is not supported.

Action: Check the globalization guide for a character set that supports binary sort.

GDK-03006 The comparison strength level setting is invalid.

Cause: An invalid comparison strength level was specified.

Action: Choose a valid comparison strength level from the list. The levels are PRIMARY, SECONDARY or TERTIARY.

GDK-03007 The composition level setting is invalid.

Cause: An invalid composition level setting was specified.

Action: Choose a valid composition level from the list. The levels are NO_ COMPOSITION or CANONICAL_COMPOSITION.

GDK-04001 Cannot map Oracle character to Unicode.

Cause: The program attempted to use a character in the Oracle character set that cannot be mapped to Unicode.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

GDK-04002 Cannot map Unicode to Oracle character.

Cause: The program attempted to use an Unicode character that cannot be mapped to a character in the Oracle character set.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

GDK-05000 A literal in the date format is too large.

Cause: The specified string literal in the date format was too long.

Action: Use a shorter string literal in the date format.

GDK-05001 The date format is too long for internal buffer..

Cause: The date format pattern was too long.

Action: Use a shorter date format pattern.

GDK-05002 The Julian date is out of range.

Cause: An illegal date range was specified.

Action: Make sure that date is in the specified range 0 - 3439760.

GDK-05003 Failure in retrieving date/time.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05010 Duplicate format code found.

Cause: The same format code was used more than once in the format pattern.

Action: Remove the redundant format code.

GDK-05011 The Julian date precludes the use of the day of the year.

Cause: Both the Julian date and the day of the year were specified.

Action: Remove either the Julian date or the day of the year.

GDK-05012 The year may only be specified once.

Cause: The year format code appeared more than once.

Action: Remove the redundant year format code.

GDK-05013 The hour may only be specified once.

Cause: The hour format code appeared more than once.

Action: Remove the redundant hour format code.

GDK-05014 The AM/PM conflicts with the use of A.M./P.M.

Cause: AM / PM was specified along with A.M. / P.M.

Action: Use either AM / PM or A.M. / P.M, but do not use both.

GDK-05015 The BC/AD conflicts with the use of B.C./A.D.

Cause: BC / AD was specified along with B.C. / A.D.

Action: Use either BC / AD or B.C. / A.D., but do not use both.

GDK-05016 Duplicate month found.

Cause: The month format code appeared more than once.

Action: Remove the redundant month format code.

GDK-05017 The day of the week may only be specified once.

Cause: The day of the week format code appeared more than once.

Action: Remove the redundant day of the week format code.

GDK-05018 The HH24 precludes the use of meridian indicator.

Cause: HH24 was specified along with the meridian indicator.

Action: Use either the HH24, or the HH12 with the meridian indicator.

GDK-05019 The signed year precludes the use of BC/AD.

Cause: The signed year was specified along with BC / AD.

Action: Use either the signed year, or the unsigned year with BC / AD.

GDK-05020 A format code cannot appear in a date input format.

Cause: A format code appeared in a date input format.

Action: Remove the format code.

GDK-05021 Date format not recognized.

Cause: An unsupported format code was specified.

Action: Correct the format code.

GDK-05022 The era format code is not valid with this calendar.

Cause: An invalid era format code was specified for the calendar.

Action: Remove the era format code or use anther calendar that supports the era.

GDK-05030 The date format pattern ends before converting entire input string.

Cause: An incomplete date format pattern was specified.

Action: Rewrite the format pattern to cover the entire input string.

GDK-05031 The year conflicts with the Julian date.

Cause: An incompatible year was specified for the Julian date.

Action: Make sure that the Julian date and the year are not in conflict.

GDK-05032 The day of the year conflicts with the Julian date.

Cause: An incompatible day of year was specified for the Julian date.

Action: Make sure that the Julian date and the day of the year are not in conflict.

GDK-05033 The month conflicts with the Julian date.

Cause: An incompatible month was specified for the Julian date.

Action: Make sure that the Julian date and the month are not in conflict.

GDK-05034 The day of the month conflicts with the Julian date.

Cause: An incompatible day of the month was specified for the Julian date.

Action: Make sure that the Julian date and the day of the month are not in conflict.

GDK-05035 The day of the week conflicts with the Julian date.

Cause: An incompatible day of the week was specified for the Julian date.

Action: Make sure that the Julian date and the day of week are not in conflict.

GDK-05036 The hour conflicts with the seconds in the day.

Cause: The specified hour and the seconds in the day were not compatible.

Action: Make sure the hour and the seconds in the day are not in conflict.

GDK-05037 The minutes of the hour conflicts with the seconds in the day.

Cause: The specified minutes of the hour and the seconds in the day were not compatible.

Action: Make sure the minutes of the hour and the seconds in the day are not in conflict.

GDK-05038 The seconds of the minute conflicts with the seconds in the day.

Cause: The specified seconds of the minute and the seconds in the day were not compatible.

Action: Make sure the seconds of the minute and the seconds in the day are not in conflict.

GDK-05039 Date not valid for the month specified.

Cause: An illegal date for the month was specified.

Action: Check the date range for the month.

GDK-05040 Input value not long enough for the date format.

Cause: Too many format codes were specified.

Action: Remove unused format codes or specify a longer value.

GDK-05041 A full year must be between -4713 and +9999, and not be 0.

Cause: An illegal year was specified.

Action: Specify the year in the specified range.

GDK-05042 A quarter must be between 1 and 4.

Cause: Cause: An illegal quarter was specified.

Action: Action: Make sure that the quarter is in the specified range.

GDK-05043 Not a valid month.

Cause: An illegal month was specified.

Action: Make sure that the month is between 1 and 12 or has a valid month name.

GDK-05044 The week of the year must be between 1 and 52.

Cause: An illegal week of the year was specified.

Action: Make sure that the week of the year is in the specified range.

GDK-05045 The week of the month must be between 1 and 5.

Cause: An illegal week of the month was specified.

Action: Make sure that the week of the month is in the specified range.

GDK-05046 Not a valid day of the week.

Cause: An illegal day of the week was specified.

Action: Make sure that the day of the week is between 1 and 7 or has a valid day name.

GDK-05047 A day of the month must be between 1 and the last day of the month.

Cause: An illegal day of the month was specified.

Action: Make sure that the day of the month is in the specified range.

GDK-05048 A day of year must be between 1 and 365 (366 for leap year).

Cause: An illegal day of the year was specified.

Action: Make sure that the day of the year is in the specified range.

GDK-05049 An hour must be between 1 and 12.

Cause: An illegal hour was specified.

Action: Make sure that the hour is in the specified range.

GDK-05050 An hour must be between 0 and 23.

Cause: An illegal hour was specified.

Action: Make sure that the hour is in the specified range.

GDK-05051 A minute must be between 0 and 59.

Cause: Cause: An illegal minute was specified.

Action: Action: Make sure the minute is in the specified range.

GDK-05052 A second must be between 0 and 59.

Cause: An illegal second was specified.

Action: Make sure the second is in the specified range.

GDK-05053 A second in the day must be between 0 and 86399.

Cause: An illegal second in the day was specified.

Action: Make sure second in the day is in the specified range.

GDK-05054 The Julian date must be between 1 and 5373484.

Cause: An illegal Julian date was specified.

Action: Make sure that the Julian date is in the specified range.

GDK-05055 Missing AM/A.M. or PM/P.M.

Cause: Neither AM/A.M. nor PM/P.M. was specified in the format pattern.

Action: Specify either AM/A.M. or PM/P.M.

GDK-05056 Missing BC/B.C. or AD/A.D.

Cause: Neither BC/B.C. nor AD/A.D. was specified in the format pattern.

Action: Specify either BC/B.C. or AD/A.D.

GDK-05057 Not a valid time zone.

Cause: An illegal time zone was specified.

Action: Specify a valid time zone.

GDK-05058 Non-numeric character found.

Cause: A non-numeric character was found where a numeric character was expected.

Action: Make sure that the character is a numeric character.

GDK-05059 Non-alphabetic character found.

Cause: A non-alphabetic character was found where an alphabetic was expected.

Action: Make sure that the character is an alphabetic character.

GDK-05060 The week of the year must be between 1 and 53.

Cause: An illegal week of the year was specified.

Action: Make sure that the week of the year is in the specified range.

GDK-05061 The literal does not match the format string.

Cause: The string literals in the input were not the same length as the literals in the format pattern (with the exception of the leading whitespace).

Action: Correct the format pattern to match the literal. If the "FX" modifier has been toggled on, the literal must match exactly, with no extra whitespace.

GDK-05062 The numeric value does not match the length of the format item.

Cause: The numeric value did not match the length of the format item.

Action: Correct the input date or turn off the FX or FM format modifier. When the FX and FM format codes are specified for an input date, then the number of digits must be exactly the number specified by the format code. For example, 9 will not match the format code DD but 09 will match.

GDK-05063 The year is not supported for the current calendar.

Cause: An unsupported year for the current calendar was specified.

Action: Refer to the globalization guide to find out what years are supported for the current calendar.

GDK-05064 The date is out of range for the calendar.

Cause: The specified date was out of range for the calendar.

Action: Specify a date that is legal for the calendar.

GDK-05065 Invalid era.

Cause: An illegal era was specified.

Action: Make sure that the era is valid.

GDK-05066 The datetime class is invalid.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05067 The interval is invalid.

Cause: An invalid interval was specified.

Action: Specify a valid interval.

GDK-05068 The leading precision of the interval is too small.

Cause: The specified leading precision of the interval was too small to store the interval.

Action: Increase the leading precision of the interval or specify an interval with a smaller leading precision.

GDK-05069 Reserved for future use.

Cause: Reserved.

Action: Reserved.

GDK-05070 The specified intervals and datetimes were not mutually comparable.

Cause: The specified intervals and date times were not mutually comparable.

Action: Specify a pair of intervals or date times that are mutually comparable.

GDK-05071 The number of seconds must be less than 60.

Cause: The specified number of seconds was greater than 59.

Action: Specify a value for the seconds to 59 or smaller.

GDK-05072 Reserved for future use.

Cause: Reserved.

Action: Reserved.

GDK-05073 The leading precision of the interval was too small.

Cause: The specified leading precision of the interval was too small to store the interval.

Action: Increase the leading precision of the interval or specify an interval with a smaller leading precision.

GDK-05074 An invalid time zone hour was specified.

Cause: The hour in the time zone must be between -12 and 13.

Action: Specify a time zone hour between -12 and 13.

GDK-05075 An invalid time zone minute was specified.

Cause: The minute in the time zone must be between 0 and 59.

Action: Specify a time zone minute between 0 and 59.

GDK-05076 An invalid year was specified.

Cause: A year must be at least -4713.

Action: Specify a year that is greater than or equal to -4713.

GDK-05077 The string is too long for the internal buffer.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05078 The specified field was not found in the datetime or interval.

Cause: The specified field was not found in the date time or interval.

Action: Make sure that the specified field is in the date time or interval.

GDK-05079 An invalid hh25 field was specified.

Cause: The hh25 field must be between 0 and 24.

Action: Specify an hh25 field between 0 and 24.

GDK-05080 An invalid fractional second was specified.

Cause: The fractional second must be between 0 and 999999999.

Action: Specify a value for fractional second between 0 and 999999999.

GDK-05081 An invalid time zone region ID was specified.

Cause: The time zone region ID specified was invalid.

Action: Contact Oracle Support Services.

GDK-05082 Time zone region name not found.

Cause: The specified region name cannot be found.

Action: Contact Oracle Support Services.

GDK-05083 Reserved for future use.

Cause: Reserved.

Action: Reserved.

GDK-05084 Internal formatting error.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05085 Invalid object type.

Cause: An illegal object type was specified.

Action: Use a supported object type.

GDK-05086 Invalid date format style.

Cause: An illegal format style was specified.

Action: Choose a valid format style.

GDK-05087 A null format pattern was specified.

Cause: The format pattern cannot be null.

Action: Provide a valid format pattern.

GDK-05088 Invalid number format model.

Cause: An illegal number format code was specified.

Action: Correct the number format code.

GDK-05089 Invalid number.

Cause: An invalid number was specified.

Action: Correct the input.

GDK-05090 Reserved for future use.

Cause: Reserved.

Action: Reserved.

GDK-0509 Datetime/interval internal error.

Cause: This is an internal error.

Action: Contact Oracle Support Services.

GDK-05098 Too many precision specifiers.

Cause: Extra data was found in the date format pattern while the program attempted to truncate or round dates.

Action: Check the syntax of the date format pattern.

GDK-05099 Bad precision specifier.

Cause: An illegal precision specifier was specified.

Action: Use a valid precision specifier.

GDK-05200 Missing WE8ISO8859P1 data file.

Cause: The character set data file for WE8ISO8859P1 was not installed.

Action: Make sure the GDK jar files are installed properly in the Java application.

GDK-05201 Failed to convert to a hexadecimal value.

Cause: An invalid hexadecimal string was included in the HTML/XML data.

Action: Make sure the string includes the hexadecimal character in the form of &x[0-9A-Fa-f]+;.

GDK-05202 Failed to convert to a decimal value.

Cause: An invalid decimal string was found in the HTML/XML data.

Action: Make sure the string includes the decimal character in the form of &[0-9]+;.

GDK-05203 Unregistered character entity.

Cause: An invalid character entity was found in the HTML/XML data.

Action: Use a valid character entity value in HTML/XML data. See HTML/XML standards for the registered character entities.

GDK-05204 Invalid Quoted-Printable value.

Cause: An invalid Quoted-Printable data was found in the data.

Action: Make sure the input data has been encoded in the proper Quoted-Printable form.

GDK-05205 Invalid MIME header format.

Cause: An invalid MIME header format was specified.

Action: Check RFC 2047 for the MIME header format. Make sure the input data conforms to the format.

GDK-05206 Invalid numeric string.

Cause: An invalid character in the form of %FF was found when a URL was being decoded.

Action: Make sure the input URL string is valid and has been encoded correctly; %FF needs to be a valid hex number.

GDK-05207 Invalid class of the object, key, in the user-defined locale to charset mapping."

Cause: The class of key object in the user-defined locale to character set mapping table was not java.util.Locale.

Action: When you construct the Map object for the user-defined locale to character set mapping table, specify java.util.Locale for the key object.

GDK-05208 Invalid class of the object, value, in the user-defined locale to charset mapping.

Cause: The class of value object in the user-defined locale to character set mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined locale to character set mapping table, specify java.lang.String for the value object.

GDK-05209 Invalid rewrite rule.

Cause: An invalid regular expression was specified for the match pattern in the rewrite rule.

Action: Make sure the match pattern for the rewriting rule uses a valid regular expression.

GDK-05210 Invalid character set.

Cause: An invalid character set name was specified.

Action: Specify a valid character set name.

GDK-0521 Default locale not defined as a supported locale.

Cause: The default application locale was not included in the supported locale list.

Action: Include the default application locale in the supported locale list or change the default locale to the one that is in the list of the supported locales.

GDK-05212 The rewriting rule must be a String array with three elements.

Cause: The rewriting rule parameter was not a String array with three elements.

Action: Make sure the rewriting rule parameter is a String array with three elements. The first element represents the match pattern in the regular expression, the second element represents the result pattern in the form specified in the JavaDoc of ServletHelper.rewriteURL, and the third element represents the Boolean value "True" or "False" that specifies whether the locale fallback operation is performed or not.

GDK-05213 Invalid type for the class of the object, key, in the user-defined parameter name mapping.

Cause: The class of key object in the user-defined parameter name mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined parameter name mapping table, specify java.lang.String for the key object.

GDK-05214 The class of the object, value, in the user-defined parameter name mapping, must be of type \"java.lang.String\."

Cause: The class of value object in the user-defined parameter name mapping table was not java.lang.String.

Action: When you construct the Map object for the user-defined parameter name mapping table, specify java.lang.String for the value object.

GDK-05215 Parameter name must be in the form [a-z][a-z0-9]*.

Cause: An invalid character was included in the parameter name.

Action: Make sure the parameter name is in the form of [a-z][a-z0-9]*.

GDK-05216 The attribute \"var\" must be specified if the attribute \"scope\" is set.

Cause: Despite the attribute "scope" being set in the tag, the attribute "var" was not specified.

Action: Specify the attribute "var" for the name of variable.

GDK-05217 The \"param\" tag must be nested inside a \"message\" tag.

Cause: The "param" tag was not nested inside a "message" tag.

Action: Make sure the tag "param" is inside the tag "message".

GDK-05218 Invalid \"scope\" attribute is specified.

Cause: An invalid "scope" value was specified.

Action: Specify a valid scope as either "application," "session," "request," or "page".

GDK-05219 Invalid date format style.

Cause: The specified date format style was invalid.

Action: Specify a valid date format style as either "default," "short," or "long"

GDK-05220 No corresponding Oracle character set exists for the IANA character set.

Cause: An unsupported IANA character set name was specified.

Action: Specify the IANA character set that has a corresponding Oracle character set.

GDK-05221 Invalid parameter name.

Cause: An invalid parameter name was specified in the user-defined parameter mapping table.

Action: Make sure the specified parameter name is supported. To get the list of supported parameter names, call LocaleSource.Parameter.toArray.

GDK-05222 Invalid type for the class of the object, key, in the user-defined message bundle mapping.

Cause: The class of key object in the user-defined message bundle mapping table was not "java.lang.String."

Action: When you construct the Map object for the user-defined message bundle mapping table, specify java.lang.String for the key object.

GDK-05223 Invalid type for the class of the object, value, in the user-defined message bundle mapping.

Cause: The class of value object in the user-defined message bundle mapping table was not "java.lang.String."

Action: When you construct the Map object for the user-defined message bundle mapping table, specify java.lang.String for the value object.

GDK-05224 Invalid locale string.

Cause: An invalid character was included in the specified ISO locale names in the GDK application configuration file.

Action: Make sure the ISO locale names include only valid characters. A typical name format is an ISO 639 language followed by an ISO 3166 country connected by a dash character; for example, "en-US" is used to specify the locale for American English in the United States.

GDK-06001 LCSDetector profile not available.

Cause: The specified profile was not found.

Action: Make sure the GDK jar files are installed properly in the Java application.

GDK-06002 Invalid IANA character set name or no corresponding Oracle name found.

Cause: The IANA character set specified was either invalid or did not have a corresponding Oracle character set.

Action: Check that the IANA character is valid and make sure that it has a corresponding Oracle character set.

GDK-06003 Invalid ISO language name or no corresponding Oracle name found.

Cause: The ISO language specified was either invalid or did not have a corresponding Oracle language.

Action: Check to see that the ISO language specified is either invalid or does not have a corresponding Oracle language.

GDK-06004 A character set filter and a language filter cannot be set at the same time.

Cause: A character set filter and a language filter were set at the same time in a LCSDetector object.

Action: Set only one of the two -- character set or language.

GDK-06005 Reset is necessary before LCSDetector can work with a different data source.

Cause: The reset method was not invoked before a different type of data source was used for a LCSDetector object.

Action: Call LCSDetector.reset to reset the detector before switching to detect other types of data source.

ORA-17154 Cannot map Oracle character to Unicode.

Cause: The Oracle character was either invalid or incomplete and could not be mapped to an Unicode value.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

ORA-17155 Cannot map Unicode to Oracle character.

Cause: The Unicode character did not have a counterpart in the Oracle character set.

Action: Write a separate exception handler for the invalid character, or call the withReplacement method so that the invalid character can be replaced with a valid replacement character.

Glossary

CSS

Cascading style sheet.

character set

Defines the binary values that are associated with the characters that make up a language. For example, you can use the ISO-8859-1 character set to encode most Western European languages.

database access descriptor (DAD)

Describes the connect string and Oracle parameters of a target database to which an Oracle HTTP Server `mod_plsql` module connects.

encoding

The character set used in a particular programming environment for the locale to which an Internet application is serving. *See* **page encoding**, **character set**.

JSP

JavaServer Page. An extension to servlet functionality that provides a simple programmatic interface to Web pages. JSPs are HTML pages with special tags and embedded Java code that is executed on the Web or application server. JSPs provide dynamic functionality to HTML pages. They are actually compiled into servlets when first requested and run in the servlet container.

locale

Refers to a language and the region (territory) in which the language is spoken. Information about the region includes formats for dates and currency.

MIME

Multipurpose Internet Mail Extensions. A mail type that defines the message structure for different 8-bit character sets and multi-part messages.

monolingual Internet application

Each instance in the application supports a different locale. Users invoke the instance that serves their locale.

multilingual Internet application

One instance supports several locales. All users invoke the same instance regardless of locale.

page encoding

The character set an HTML page uses for the locale to which an Internet application is serving.

PSP

PL/SQL Server Pages

Unicode

A universal character set that defines binary values for characters in almost all languages. Unicode characters can be encoded in 1 to 4 bytes in the UTF-8 character set, in 2 to 4 bytes in the UTF-16 character set, and in 4 bytes in the UTF-32 character set.

Index

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W