Internationalizing and Localizing Web Applications

The process of preparing an application to support more than one language and data format is called internationalization.

Localization is the process of adapting an internationalized application to support a specific region or locale.

Examples of locale-dependent information include messages and user interface labels, character sets and encoding, and date and currency formats.

Although all client user interfaces should be internationalized and localized, it is particularly important for web applications because of the global nature of the web.

The following topics are addressed here:

- Java Platform Localization Classes
- Providing Localized Messages and Labels
- Date and Number Formatting
- Character Sets and Encodings

Java Platform Localization Classes

In the Java platform, java.util.Locale represents a specific geographical, political, or cultural region.

The string representation of a locale consists of the international standard two-character abbreviation for language and country and an optional variant, all separated by underscore (_) characters.

Examples of locale strings include `fr` (French), `de_CH` (Swiss German), and `en_US_POSIX` (English on a POSIX-compliant platform).

Locale-sensitive data is stored in a java.util.ResourceBundle.

A resource bundle contains key-value pairs, where the keys uniquely identify a locale-specific object in the bundle.

A resource bundle can be backed by a text file (properties resource bundle) or a class (list resource bundle) containing the pairs.

You construct resource bundle instance by appending a locale string representation to a base name.

The Duke's Tutoring application contains resource bundles with the base name `messages.properties` for the locales `pt` (Portuguese), `de` (German),`es` (Spanish), and `zh` (Chinese).

The default locale, `en` (English), which is specified in the `faces-config.xml` file, uses the resource bundle with the base name, `messages.properties`.

For more details on internationalization and localization in the Java platform, see .

Providing Localized Messages and Labels

Messages and labels should be tailored according to the conventions of a user's language and region.

There are two approaches to providing localized messages and labels in a web application:

- Provide a version of the web page in each of the target locales and have a controller servlet dispatch the request to the appropriate page depending on the requested locale.

This approach is useful if large amounts of data on a page or an entire web application need to be internationalized.

- Isolate any locale-sensitive data on a page into resource bundles, and access the data so that the corresponding translated message is fetched automatically and inserted into the page.

Thus, instead of creating strings directly in your code, you create a resource bundle that contains translations and read the translations from that bundle using the corresponding key.

The Duke's Tutoring application follows the second approach.

Here are a few lines from the default resource bundle `messages.properties`:

```
nav.main=Main page
nav.status=View status
nav.current_session=
View current tutoring session
nav.park=View students at the park
```

```
nav.admin=Administration
admin.nav.main=
Administration main page
admin.nav.create_student=
Create new student
admin.nav.edit_student=Edit student
admin.nav.create_guardian=
Create new guardian
admin.nav.edit_guardian=
Edit guardian
```

```
admin.nav.create_address=
Create new address
admin.nav.edit_address=Edit address
admin.nav.activate_student=
Activate student
```

Establishing the Locale

To get the correct strings for a given user, a web application either retrieves the locale (set by a browser language preference) from the request using the `getLocale` method, or allows the user to explicitly select the locale.

A component can explicitly set the locale by using the `fmt:setLocale` tag.

The `locale-config` element in the configuration file registers the default locale and also registers other supported locales.

This element in Duke's Tutoring registers English as the default locale and indicates that German, Spanish, Portuguese, and Chinese are supported locales.

```
<locale-config>
<default-locale>en</default-locale>
<supported-locale>
de
</supported-locale>
```

```
<supported-locale>
es
</supported-locale>
<supported-locale>
pt
</supported-locale>
<supported-locale>
zh
</supported-locale>
</locale-config>
```

The Status Manager in the Duke's Tutoring application uses the `getLocale` method to retrieve the locale and a `toString` method to return a localized translation of a student's status based on the locale.

```
public class StatusManager {
private FacesContext ctx =
FacesContext.getCurrentInstance();
private Locale locale;
// Creates a new instance
```

```
// of StatusManager
public StatusManager() {
locale =
ctx.getViewRoot().getLocale();
}
public String getLocalizedStatus
(StatusType status)
{ return status.toString(locale); }
}
```

Setting the Resource Bundle

The resource bundle is set with the `resource-bundle` element in the configuration file.

The setting for Duke's Tutoring looks like this:

```
<resource-bundle>
<base-name>
dukestutoring.web.messages.Messages
</base-name>
<var>bundle</var>
</resource-bundle>
```

After the locale is set, the controller of a web application could retrieve the resource bundle for that locale and save it as a session attribute (see Associating Objects with a Session) for use by other components or simply be used to return a text string appropriate for the selected locale:

```
public String toString
 (Locale locale) {

```

```
ResourceBundle res =
ResourceBundle.getBundle
("dukestutoring.web.messages.Messages",
locale);
return res.getString(name() +
".string");
}
```

Alternatively, an application could use the `f:loadBundle` tag to set the resource bundle.

This tag loads the correct resource bundle according to the locale stored in FacesContext.

```
<f:loadBundle
basename=
"dukestutoring.web.messages.Messages"
var="bundle"
/>
```

Resource bundles containing messages that are explicitly referenced from a JavaServer Faces tag attribute using a value expression must be registered using the `resource-bundle` element of the configuration file.

For more information on using this element, see Registering Custom Localized Static Text.

Retrieving Localized Messages

A web component written in the Java programming language retrieves the resource bundle from the session:

```
ResourceBundle messages =
 (ResourceBundle)
session.getAttribute("messages");
```

Then it looks up the string associated with the key `person.lastName` as follows:

```
messages.
getString("person.lastName");
```

You can only use a `message` or `messages` tag to display messages that are queued onto a component as a result of a converter or validator being registered on the component.

The following example shows a `message` tag that displays the error message queued on the `userNo` input component if the validator registered on the component fails to validate the value the user enters into the component.

```
<h:inputText id="userNo"
value="#{UserNumberBean.userNumber}
">

<f:validateLongRange
```

```
minimum="0" maximum="10"
/>
...
<h:message
style="
color: red;
text-decoration: overline"
id="errors1" for="userNo"
/>
```

For more information on using the `message` or `messages` tags, see Displaying Error Messages with the `h:message` and `h:messages` Tags.

Messages that are not queued on a component and are therefore not loaded automatically are referenced using a value expression.

You can reference a localized message from almost any JavaServer Faces tag attribute.

The value expression that references a message has the same notation whether you loaded the resource bundle with the `loadBundle` tag or registered it with the `resource-bundle` element in the configuration file.

The value expression notation is `var.message`, in which `var` matches the `var` attribute of the `loadBundle` tag or the `var` element defined in the `resource-bundle` element of the configuration file, and `message` matches the key of the message contained in the resource bundle, referred to by the `var` attribute.

Here is an example from `editAddress.xhtml` in Duke's Tutoring:

```
<h:outputLabel for="country"
value=
"#{bundle['address.country']}:"
/>
```

Notice that `bundle` matches the `var` element from the configuration file and that `country` matches the key in the resource bundle.

For information on using localized messages in JavaServer Faces applications, see Displaying Components for Selecting Multiple Values.

Date and Number Formatting

Java programs use the `DateFormat.getDateInstance(int, locale)` to parse and format dates in a locale-sensitive manner.

Java programs use the `NumberFormat.getXXXInstance(locale)` method, where *XXX* can be `Currency`, `Number`, or `Percent`, to parse and format numerical values in a locale-sensitive manner.

An application can use date/time and number converters to format dates and numbers in a locale-sensitive manner.

For example, a shipping date could be converted as follows:

```
<h:outputText
value="#{cashier.shipDate}">
<f:convertDateTime
dateStyle="full"/>
</h:outputText>
```

For information on JavaServer Faces converters, see Using the Standard Converters.

Character Sets and Encodings

The following sections describe character sets and character encodings.

Character Sets

A character set is a set of textual and graphic symbols, each of which is mapped to a set of nonnegative integers.

The first character set used in computing was US-ASCII.

It is limited in that it can represent only American English.

US-ASCII contains uppercase and lowercase Latin alphabets, numerals, punctuation, a set of control codes, and a few miscellaneous symbols.

Unicode defines a standardized, universal character set that can be extended to accommodate additions.

When the Java program source file encoding doesn't support Unicode, you can represent Unicode characters as escape sequences by using the notation \u*XXXX*, where *XXXX* is the character's 16-bit representation in hexadecimal.

For example, the Spanish version of the Duke's Tutoring message file uses Unicode for non-ASCII characters:

```
nav.main=P\u00e1gina Principal
nav.status=Mirar el estado
nav.current_session=
Ver sesi\u00f3n actual del tutorial
nav.park=
Ver estudiantes en el Parque
nav.admin=Administraci\u00f3n
admin.nav.main=P\u00e1gina
principal de administraci\u00f3n
admin.nav.create_student=
Crear un nuevo estudiante
```

```
admin.nav.edit_student=Editar
informaci\u00f3n del estudiante
admin.nav.create_guardian=
Crear un nuevo guardia
admin.nav.edit_guardian=
Editar guardia
admin.nav.create_address=
Crear una nueva direcci\u00f3n
admin.nav.edit_address=
Editar direcci\u00f3n
```

```
admin.nav.activate_student=
Activar estudiante
```

Character Encoding

A character encoding maps a character set to units of a specific width and defines byte serialization and ordering rules.

Many character sets have more than one encoding.

For example, Java programs can represent Japanese character sets using the `EUC-JP` or `Shift-JIS` encodings, among others.

Each encoding has rules for representing and serializing a character set.

The ISO 8859 series defines 13 character encodings that can represent texts in dozens of languages.

Each ISO 8859 character encoding can have up to 256 characters.

ISO-8859-1 (Latin-1) comprises the ASCII character set, characters with diacritics (accents, diaereses, cedillas, circumflexes, and so on), and additional symbols.

UTF-8 (Unicode Transformation Format, 8-bit form) is a variable-width character encoding that encodes 16-bit Unicode characters as one to four bytes.

A byte in UTF-8 is equivalent to 7-bit ASCII if its high-order bit is zero; otherwise, the character comprises a variable number of bytes.

UTF-8 is compatible with the majority of existing web content and provides access to the Unicode character set.

Current versions of browsers and email clients support UTF-8.

In addition, many new web standards specify UTF-8 as their character encoding.

For example, UTF-8 is one of the two required encodings for XML documents (the other is UTF-16).

Web components usually use `PrintWriter` to produce responses; `PrintWriter` automatically encodes using ISO-8859-1.

Servlets can also output binary data using `OutputStream` classes, which perform no encoding.

An application that uses a character set that cannot use the default encoding must explicitly set a different encoding.