9

Internationalization and Localization

CERTIFICATION OBJECTIVES

Applications often need the flexibility to support the language and presentation customs for several geographic locations. In Java parlance, this process is known as internationalization and localization. This chapter covers the issues surrounding this process and the aspects of an application affected by it.

CERTIFICATION OBJECTIVE 9.01

State Three Aspects of Any Application That Might Need to Be Varied or Customized in Different Deployment Locales

Internationalization is the process of preparing application code to support multiple languages, and *localization* is the process of adapting an internationalized application so that it supports a specific language or locale. A *locale* is an environment that includes regional and language-specific information.

Internationalization and Localization

Internationalization involves isolating portions of the application that present output data to the user so that the data can be converted to the appropriate language and character set. Localization involves translating these strings into a specific language and maintaining them in a file that the application can access—for example, a property file. Thus, internationalizing an application allows it to be adapted to new languages and regions, while localization provides the adaptation of an internationalized application to a specific country or region. It is important to note that the Enterprise JavaBeans (EJB) container need not be running in the same locale as the client browser.

Applications need to customize the presentation of data according to the locale of the user. An application must be internationalized, and then it can be localized. During internationalization (also known as *I18N*, because the number of characters between the first and last character is 18), locale dependencies are separated from an application's source code. Examples of these locale dependencies include user interface labels; messages character set; encoding; and numeric, currency, and time formats. During localization (also known as *L10N*), an internationalized application is adapted to a specific locale. Internationalization and localization make Java Enterprise Edition (JEE) applications available to a global audience.

Internationalization is typically overlooked when developing an enterprise web application, because these sorts of applications are usually targeted to a particular local user space. When developing an enterprise application that may be used globally, however, you should consider internationalization from the outset. It is easier to design an application that is capable of being internationalized than to redesign an existing application later. As with other redesigns, a great deal of time and money can be saved by planning for internationalization and localization at the outset.

With a web-based enterprise application, the presentation layer is the focus of internationalization and localization efforts. The presentation layer includes JavaServer Pages (JSPs), servlets, and any supporting helper JavaBeans components.

Overview of Internationalizing an Application

Before we get to the details of internationalizing an application, let's review our objectives. After the architectural design and development is completed, an internationalized enterprise application will have the following characteristics:

- With the addition of localization data, the same executable—such as an application Enterprise Archive (a file with an *.ear* extension)—can run worldwide.
- GUI component labels and other textual elements (such as messages) are not hard-coded within the program but are stored outside and retrieved dynamically.
- Regionally dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- Recompilation is not required to support a new language. It can be localized quickly by the addition of the new language property file entries.

So what should you analyze to internationalize your application? Many types of data vary with region or language, so your approach depends on the application being delivered. A nonexhaustive item list of this data includes the following:

messages	labels on GUI components	online help
colors	graphics	Icons
dates	times	Numbers
currencies	measurements	phone numbers
personal titles	postal addresses	page layouts
legal rules	sounds	

CERTIFICATION OBJECTIVE 9.02

List Three Features of the Java Programming Language That Can Be Used to Create an Internationalizable/Localizable Application

Now let's take a look at Java's support for internationalization and localization. We'll look at specific API classes and objects that have been designed to help with *I18N* and *L10N*.

Java Support for Internationalization and Localization

An internationalized JEE application cannot assume that it is being executed from a single locale and often needs to service requests for many locales simultaneously. That is to say that a client request will arrive with an associated locale and consequently expect the response with the same locale. Because internationalization affects all tiers of a JEE application, it is an architecturally fundamental issue. Unfortunately, on many JEE projects, application internationalization is an afterthought and usually requires a great deal of refactoring to incorporate it later. As stated previously, internationalization and localization dependencies need to be identified during the project design phase.

Let's review some of the internationalization and localization classes, tools, and features available to use in Java.

Using java.util.Properties for Localization

The *java.util.Properties* class represents a set of properties that can be persisted. The properties can be loaded from or saved to a stream. Both the key and its corresponding lookup value in the list of properties is a string. The properties object typically stores information about the characteristics of an application or its environment, and this can also include information pertaining to internationalization and localization. By creating a properties object and using the `load()` method, a program can read a localized properties file or any arbitrary input stream and then access the appropriate localized values using the same key:

```
Properties props = new Properties();
String myProps = "l10nfile";
props.load(new BufferedInputStream(new FileInputStream(myProps);
String msgvalue = System.getProperty("msgkey");
```

See "ResourceBundle" a little later in this chapter for more advanced uses of properties for localization.

Locale

As mentioned, a locale is a way of identifying and using the exact language and cultural settings for a particular session or user. In Java, a locale is identified by one, two, and occasionally three elements:

- **Language** This is the basic identifier for a locale. It contains a valid International Standards Organization, ISO 639, two-letter language code. Examples are *en* for English and *es* for Spanish. (A complete list of two-letter language codes can be found at *http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt*.)
- **Regional variation** This is a country code. It contains a valid ISO 3166 two-letter country code. Examples are *GB* for United Kingdom, *CO* for Colombia, and *US* for United States. (A complete list can be found at *http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html or http://std.dkuug.dk/i18n/ISO_3166*.)
- **Variant** This element is less frequently specified. It is used for creating locales with vendor- or browser-specific code. Examples are *WIN* for Windows, *MAC* for Macintosh, and *POSIX* for POSIX (Linux or UNIX). It is also used to allow for the possibility of more than one locale per country and language combination. Most European countries also now have the *EURO* variant for currency formatting.

Locales are defined with the language and country code separated by an underscore, like so: *es_CO* or *en_US*. The *Locale* class provides a number of constants that you can use to create locale objects for common locales. For example, *Locale.US* creates a locale object for the United States. Other locale names include, for example, *de* for German, *de_CH* for Swiss-style German, and *de_CH_POSIX* for Swiss-style German on a POSIX-compliant platform.

The locale object controls formatting for numeric, data, currency, and percent display. It can affect many other areas, such as how case folding (uppercasing and lowercasing of letters) is handled. It can affect the way a list is sorted (called the *collation sequence*), or which day appears in the leftmost column on a calendar. Based on the locale, Java provides mechanisms for loading the user interface, messages, and specialized code from *resource bundles* (which are defined next). In short, locales provide a way of configuring classes to match the user requirements dynamically.

Platforms other than Java have slight variations for locale objects, names, and structures. For example, Microsoft Windows uses a proprietary three-letter code to identify a locale.

Many developers confuse character-set problems with locales. Character-set problems are usually the primary aspect that is addressed when internationalizing code. Terms such as *double-byte enabling*, *kanji*, or *Unicode enabling* are key internationalization discussions. However, the issues surrounding character set are only a part of making a product locale-aware. Without the correct character-set handling, data will not display correctly. However, internationalization and localization go beyond just manipulating the characters.

In Java, the `java.util.Locale` object represents a specific geographical or cultural region. An operation that requires a locale to perform its task is called *locale-sensitive* and uses the locale to refine information for the user. For example, displaying a number is a locale-sensitive operation—that is, the number should be formatted according to the conventions of the user's native country or region.

Because locale objects are merely region identifiers, no validity check is performed when they are constructed. You can query particular resources to determine whether they are available for your locale. For example, you can call the `getAvailableLocales()` method on *DateFormat* to obtain an array of locales for which *DateFormat*s are installed. When a resource is requested for a particular locale, the best available match is returned, which is not necessarily precisely what was requested.

After you've created a locale, you can access it for information about itself. Use `getCountry()` for the ISO country code and `getLanguage()` for the ISO language code. You can use `getDisplayCountry()` for the name of the country suitable for displaying to the user. Similarly, you can use `getDisplayLanguage()` for the name of the language suitable for displaying to the user. Interestingly, the `getDisplay` accessor methods are locale-sensitive and have two versions: one that uses the default locale and one that uses the locale specified in the argument.

ResourceBundle

The *java.util.ResourceBundle* class defines a naming convention for locales, which should be used whenever organizing resources by locale. Resource bundles hold locale-specific objects. When your class requires a locale-specific resource, for example a string, your class can load it from the resource bundle that matches the current user's locale. Correspondingly, you can write class code that is independent of the user's locale. This allows you to write classes that can do the following:

- Can be localized—translated into different languages
- Can handle multiple locales simultaneously
- Can be modified to support additional locales

A *resource bundle* is a set of related classes that are inherited from *java.util.ResourceBundle*. Each *ResourceBundle* subclass has the same base name plus a component that identifies its locale. For example, suppose your resource bundle is named *UCResources*. The first class you are likely to write is the default resource bundle, which simply has the same name as its family—*UCResources*. You can then create related locale-specific classes as needed—for example, you can provide a German class named *UCResources_de*.

Each subclass of *ResourceBundle* contains the same objects, but the objects have been translated for the locale represented by that subclass of *ResourceBundle*.

The resource bundle lookup searches for classes with a name assembled from the following that are concatenated, separated by underscores. Consider the following example: *UCResources_en_GB_cockney*. This class name includes the following:

- A base class *(UCResources)*
- The desired language *(en)*
- The desired country *(GB)*
- The desired variant *(cockney)*

During runtime, if the class or a properties file of the same name with the properties extension cannot be found, the lookup will review each of the elements in turn until a match is found. By providing a class with no suffixes (that is, the base class), a match will always be found.

The base class must also be fully qualified (for example, *UCPackage.UCResources*, not just *UCResources*). It must also be accessible by your class code; it cannot be a private class to the package where *ResourceBundle.getBundle* is called. While keys must be defined as *java.lang.String*, the lookup values can be any subclass of *java.lang.Object*. *PropertyResourceBundle* is a subclass of *ResourceBundle* that handles resources for a locale using a set of static strings from a property object containing the resource data. *ResourceBundle.getBundle* will look for the appropriate properties object and create a *PropertyResourceBundle* that refers to it.

Character Sets

A *character set* is a group of textual or graphical symbols that is mapped to a set of (positive) integers called *code points*. The ASCII (American Standard Code for Information Interchange) character set is commonly used for representing American English. ASCII contains uppercase and lowercase Roman alphabets, European numerals, punctuation, a group of control codes, and some symbols. For example, the ASCII code point for *A* is 65 (hexadecimal 41).

The ISO 8859 character-set series was created because ASCII was not good for supporting languages other than American English. Each ISO 8859 character set can have up to 256 code points. ISO 8859-1, also known as "Latin-1," has the ASCII character set, symbols, and characters with accents, circumflexes, and other diacritics. With the ISO 8859 series of character sets, it is possible to represent texts for dozens of languages.

Unicode (ISO 10646) defines a standard and universal character set. It was designed to represent practically all character sets in use around the world and can be extended. Unicode encompasses alphabetic scripts and ideographic writing systems, and it may be rendered in any direction.

Unicode is an international effort to provide a single character set for everyone. Java uses the Unicode 2.*x* character-encoding standard. In Unicode, every character uses two bytes. Ranges of character encoding represent various writing systems and other special symbols. For example, Unicode characters in the range 0x9FFF through 0xAC00 represent the Han characters used in Asia: China, Japan, Korea, Taiwan, and Vietnam. Despite the obvious advantages of Unicode, it has a big shortcoming: Unicode support is limited on many platforms because of the lack of fonts capable of displaying all the Unicode characters.

The Java programming language internally represents characters and string objects as encoded Unicode. Classes written in the Java programming language can process data in multiple languages, natively performing localized operations such as string comparison, parsing, and collation. Unicode characters in a Java class may be represented as escape sequences, using the notation *u*XXXX, where *XXXX* is the character's 16-bit code point in hexadecimal. These Unicode-escaped strings are useful for Java source files that are not encoded as Unicode.

Unicode Transformation Format (UTF), where the *U* stands for UCS (Universal Character Set), is a multibyte encoding format that stores some characters in one byte and others in two or three bytes. If most of the data is ASCII based, UTF is more compact than Unicode, but in a worst-case scenario, the UTF string can be 50 percent larger than the equivalent Unicode string. Overall, UTF is still fairly efficient and is the most widely used character-encoding scheme.

Encoding

An *encoding* will map the code points in a character set to units of a specific width, and it defines byte serialization and ordering rules. Many of these character sets have more than one encoding. The *java.io* package contains classes that support the reading and writing of character data streams using a variety of encoding schemes. Some of these classes are discussed later in this chapter, and they all have names that

end in either *Reader* (for example, *BufferedReader* and *InputStreamReader*) or *Writer* (for example, *BufferedWriter, PrintWriter,* and *OutputStreamWriter*).

Programmers use *PrintWriter* within JSPs and servlets to produce textual responses, which are automatically encoded. It is possible for servlets to output binary data with no encoding using an *OutputStream* class. You must explicitly set an encoding if you create an application that uses a character set that cannot be handled by the default encoding (ISO 8859-1, *Latin_1*).

UTF-8 is an 8-bit form of UTF, the unification of US-ASCII and Unicode. UTF-8 is a variable-width character encoding that encodes 16-bit Unicode characters into one or two bytes. Encoding internationalized content in UTF-8 is recommended by Sun because it is compatible with the majority of existing web content and provides access to the Unicode character set. In addition, most current browsers and e-mail clients support it, and it is one of the two required encoding schemes for XML documents (the other being UTF-16).

Handling Text Dates and Numbers with the java.text Package

This package provides several classes and interfaces that can be used for handling text, dates, numbers, and messages in ways that are independent of natural languages. This means that an application can be created in a language-independent manner and can rely on separate, dynamically linked, (and therefore) localized resources.

All of the classes in the *java.text* package are sensitive to either the default or provided locale. This package of classes provides the ability to format numbers, dates, and messages; to parse, search, and sort through strings; and to iterate over characters, words, sentences, and line breaks.

This package contains three main groups of classes and interfaces: *iteration, formatting,* and *string collation*. Here is a list of some of the classes in the *java.text* package:

- *Annotation*
- *CollationKey*
- *Collator*
- *Format*

An *annotation* object is a wrapper for a text attribute value if the attribute has annotation characteristics. One characteristic is the text range to which the attribute is applied; this is critical to the semantics of the range. Wrapping the

attribute value into an annotation object guarantees that adjacent text does not get merged, even if the attribute values are equal, and it indicates to text containers that the attribute should be discarded if the underlying text is modified.

The different languages of the world use alphabets that differ, and thus they require unique ways to sort strings written in those languages. *Collation* is the process of sorting strings according to locale-specific customs. The *Collator* is an abstract base class that provides locale-sensitive string comparison. Use this class when building search and sort routines for natural-language text. Subclasses implement specific collation requirements. Use the static factory method, `getInstance()`, to obtain the proper collator object for a given locale.

A *CollationKey* represents a string that is controlled by the rules of a specific collator object. Comparing two *CollationKey*s will return the relative order of the strings they represent and is typically faster than using the `Collator.compare()` method when sorting a list of Strings. *CollationKey*s are generated by calling the `getCollationKey()` method on a collator, and they can be compared only when generated from the same collator. The generation process converts the string to a series of bits that can then be compared bitwise. This translates into fast comparisons once the keys are generated. The cost of generating keys is justified in faster comparisons when strings need to be compared many times. Alternatively, the first couple of characters of each string often determine the result of a comparison. `Collator.compare()` examines only as many characters as it needs, and as soon as an inequality is arrived at, the comparison is over.

Format is obviously important. Again, Format is an abstract base class for formatting locale-sensitive information such as dates, messages, and numerics. Format defines the programming interface for formatting locale-sensitive objects into strings (use the `format()` method) and for parsing strings into objects (use the `parseObject()` method). Any string formatted by Format is parsable by `parseObject()`. Format has three concrete subclasses, *DateFormat*, *MessageFormat*, and *NumberFormat*, which will be covered later in this chapter in the sections "Internationalization with Respect to Data Handling," "Message Formatting," and "Date Formatting."

InputStreamReader

An *InputStreamReader* is a mechanism for converting from byte streams to character streams: It reads bytes and converts them into characters according to a specified character encoding. The class has two constructors: one with no arguments will use the platform default encoding, and the other takes an encoding argument (as a string). The ISO numbers are used to represent an encoding—for example, ISO 8859-9 is represented by 8859_9.

There is no simple way to determine which encodings are supported, but you can call the `getEncoding()` method to obtain the name of the encoding used by the *InputStreamReader*. Characters that do not exist in a specific character set are substituted with another character, usually a question mark.

OutputStreamWriter

An *OutputStreamWriter* is a mechanism for converting data from character streams to byte streams: Characters written to it are translated into bytes according to a specified character encoding. The encoding that it uses is either specified explicitly by name (again an ISO number), or the default encoding for the platform is the default.

Every invocation of the `write()` method will in turn call the encoding converter for the given character(s). The converted bytes are buffered before being written to the output stream.

Again, as with *InputStreamReader*, you cannot determine which encodings are supported, but calling the `getEncoding()` method will return the encoding used by the *OutputStreamWriter*. Characters that do not exist in a specific character set are substituted with another character, usually a question mark.

Internationalization with Respect to Data Handling

Data handling is the part of a web application most affected by internationalization, with impact in three areas: data input, storage, and presentation.

Data input is typically input to a web application by a Hypertext Transfer Protocol (HTTP) post back to a servlet from a form on a Hypertext Markup Language (HTML) page. Typically, the client platform will provide a means for inputting the data.

The browser running in the client's native locale encodes the form parameter data in the HTTP request so that it is in a readable format for the web application. When the application receives the data, it is in Unicode format, obviating character-set issues. Word breaking or parsing can be handled with the *BreakIterator* class in the *java.text* package.

Data storage for international applications means setting your database to a Unicode 2.0 character encoding (such as UTF-8 or UTF-16). This allows data to be saved in many different languages. The content you save must be entered properly from the web tier. The Java Database Connectivity (JDBC) drivers must support the encoding you choose.

To enable locale-independent data formatting, an application must be designed to present localized data appropriately for a target locale. The developer must ensure that locale-sensitive text such as dates, times, currency, and numbers are presented in a locale-specific way. If you design your text-related classes in a locale-independent way, they are reusable throughout an application.

The following code demonstrates methods used to format currency in locale-specific and locale-independent ways:

```
package com.ucny.utils;
...
public class Formatter {
...
//Format currency for the default locale.
public static String formatCCY(double amount) {
  String pattern = "$###,###,###.00";
  //Get number format for the default (system) locale.
  NumberFormat nf = NumberFormat.getCurrencyInstance();
  DecimalFormat df = (DecimalFormat)nf;
  df.setMinimumFractionDigits;
  df.setMaximumFractionDigits;
  df.setDecimalSeparatorAlwaysShown(true);
  df.applyPattern(pattern);
  return df.format(amount);
}
//Format currency for the specified locale.
public static String formatCCY(Locale locale, string prefix,
double amount) {
  String pattern = prefix+"###,###,###.00";
  //Get number format for the passed in locale.
  NumberFormat nf = NumberFormat.getCurrencyInstance(locale);
  DecimalFormat df = (DecimalFormat)nf;
  df.setMinimumFractionDigits;
  df.setMaximumFractionDigits;
  df.setDecimalSeparatorAlwaysShown(true);
  df.applyPattern(pattern);
  return df.format(amount);
} ...
```

In a JSP page, the following code snippet shows calls to the currency format function for the default locale and for the Great Britain locale:

```
...
<%=JSPUtil.formatCCY(Locale.UK, order.getTotal())%>
...
<%=JSPUtil.formatCCY(Locale.UK, "£", order.getTotal())%>
...
```

These JSP expressions use the two versions of the `formatCCY()` method of the *Formatter* utility. The total that is returned from the `order.getTotal()` method is a Java double primitive data type. Note that when using this code, the JSP will need to import the *java.util.Locale* and *com.ucny.utils.Formatter* classes.

Using Java Internationalization APIs in JEE Applications

Java internationalization APIs include utility classes and interfaces for externalizing application resources, formatting messages, formatting currency and decimals, representing dates and times, and collating. The next sections explain how to use J2SE internationalization APIs in JEE applications.

Message Formatting

The *java.text.MessageFormat* class provides a generic way to create concatenated message strings. It contains a pattern string that has embedded format specifiers. The `MessageFormat.format()` method then formats an array of objects using these embedded format specifiers and returns the result in a *StringBuffer*. The *MessageFormat* class is good for formatting system-level messages such as error or logging strings. Here is an example:

```
// Format the message

String pattern =
"Catalog number {0}, item code{1}: has been sent to order processing.";

MessageFormat mf = new MessageFormat(pattern);
Object[] objs =
new Object[] {new Integer(catalogNumber),new Integer(itemNumber)};
StringBuffer result = new StringBuffer();
String message = mf.format(objs,result, new FieldPosition());
```

In this code snippet, the *MessageFormat* holds the pattern and uses it to format the resultant string, substituting formatted objects (integers) in place of the embedded format specifiers (`{0}` and `{1}`). The *MessageFormat* class is very effective for internationalizing custom tags for a JSP.

Date Formatting

Typically, enterprise applications store, compare, and perform arithmetic on date values. JEE applications typically persist date and time values to a JDBC data store using *java.sql.Date*; hold them in memory using *java.util.Date*; manipulate and

interpret them using the *java.util.Calendar* class; and parse, format, and present them using the *java.text.DateFormat* class. The *java.text.DateFormat* class is an abstract class that provides a locale-sensitive API for parsing, formatting, and normalizing dates for presentation. The *java.text.SimpleDateFormat* class implements simple date and time value formatting for all supported locales.

Collation

As mentioned, collation is the process of ordering text using language- or script-specific rules, rather than using binary comparison; it is therefore locale-specific. It is possible for a character set to have more than one collating sequences. These lists of characters may be ordered numerically or lexically. For an internationalized application, the abstract class *java.text.Collator* is recommended for use when ordering lists of items. The *java.text.Collator* class and its related classes provide collation facilities that are locale-aware. For example, a component that produces ordered lists of NASDAQ stock issue entries could use *Collator* to place the entries in an order appropriate to the client's locale.

You could rely on the database to provide this collation. However, this may not be a good idea for internationalization, because most databases support a single sort order (typically specified at installation) and may not be portable to another database vendor.

Web Tier Internationalization

The web tier has JSP, JavaBean, and servlet components that need to be designed with internationalization and localization in mind. The essential areas that are covered for *I18N* and *L10N* in this layer are

- HTTP requests and responses
- Design of web tier components

HTTP Requests and Responses

For an internationalized web application to work correctly, it must be able to determine the encoding of an incoming request and then ensure that the outgoing response is encoded the same way. However, the default locale for any component in the web tier is not the calling client's locale but the actual web container's default locale. To complicate matters, in a distributed environment, the default locale may differ among containers, making the default locale the locale of the web container servicing the specific part of the request.

The management of locale and encoding can be simplified by a few recommended techniques. The first approach is to choose a single and consistent request encoding. Therefore, if all web components transmit pages using a single encoding, requests from those generated pages will remain in that encoding. As mentioned earlier in this chapter, UTF-8 is considered the best encoding choice because it provides the broadest coverage of character sets, efficient data transmission, and wide browser support. Along with this approach, it is also recommended that you add a servlet filter to compensate for components that do not or forget to set the chosen encoding. The servlet filter sets the response encoding to a single value before a servlet or JSP page receives the request. This provides a single point of control for enforcing (and changing to) the required encoding before a servlet handles the response.

The other approaches are to store the locale and encoding either in hidden variables on HTML forms or in URL parameters, or to store this information on the server in the users' session state or in a stateful session EJB.

As mentioned, the encoding of the responses from JSP pages and servlets determines not only the format of characters in the response but also the encoding of any subsequent request from the served page.

The two attributes in the page directive for a JSP control encoding are

- *contentType*
- *pageEncoding*

Note that when using these attributes, the content type and encoding will be fixed at page translation time. When using JavaServer Pages Standard Tag Library (JSTL), it is possible for a component to explicitly set the locale via the `fmt:setLocale` tag.

For a more dynamic approach, use either a custom tag or a servlet to set encoding. For a servlet, the two ways to set encoding of the response are to use the following methods on the *ServletResponse*:

- `setContentType()`
- `setLocale()`

These methods must be called before calling the `Servlet.getWriter()` method to ensure that the writer obtained is formed with the required encoding.

When using Java Server Faces (JSF) applications that support different locales you can store localized text in a resource bundle properties file. Then you inform

the application about the existence of this file by adding the following to the JSF `faces-config.xml` file:

```
<message-bundle>WebResourceBundle.properties</message-bundle>
  <locale-config>
    <default-locale>en</default-locale<
    <!--Add other locales here.-->
  </locale-config>
```

When using JavaServer Faces , it is possible to explicitly select the locale and store the locale in the `FacesContext` object.

Design of Web Tier Components

The design and implementation of JSPs, custom tags, and JavaBean helper components are affected by I18N and L10N. Any of these components that are responsible for ordering data in a collating sequence or for formatting numbers, dates, currency, or percentages need to take into account the locale to display the data in an appropriate manner for that locale.

The two most common approaches exist for localizing JSP pages are as follows:

- *Creating a JSP for each locale, with each file stored in a separate directory in the server's name space.* Typically, a servlet or servlet filter forwards each request for a JSP page to the appropriate file based on the requesting client's locale. The name of each directory uses the standard resource bundle suffix naming convention.
- *Using resource bundles with a single JSP.* You can put together sets of localized text using locale-aware custom tags. When the page is served, the custom tags grab the necessary text from the resource bundle for the current locale.

The recommendation is to use separate JSP pages for each locale when the structure of the content and display logic differs greatly between locales or when messages depend on the target locale. For all other cases, the recommendation is to use resource bundles, especially for logging and error messages or when content varies in data values and not in content structure.

Here are the advantages to creating a JSP for each locale:

- **Maximum customizability** This approach also allows for customization of the structure as well as the content for a locale.
- **Source clarity** Everything is in one place instead of being spread across the JSP file containing the structural tags and a properties file or resource bundle containing named strings.

The disadvantage to creating a JSP for each locale is that it's difficult to keep a consistent look and feel among locales. This approach requires more maintenance because separate files must be maintained consistently for multiple locales.

Here are the advantages to using resource bundles with a single JSP:

- **Easier maintenance** A change to the JSP page is reflected for all locales.
- **Consistent page structure** The JSP keeps the same structure in all locales. Only data values, message text, and the language displayed can change.
- **Easy extensibility** Defining a new resource bundle provides support for a new locale.

Here are the disadvantages to using resource bundles with a single JSP:

- Customizing its structure to locales is more difficult, because a single JSP produces the content for every locale.
- Page encoding must be compatible with the encoding of all application character sets.

Logging and Error Messages

Internationalization and localization are also needed when providing users and administrators with meaningful messages if exceptional conditions occur.

Messages and Exceptions

The presentation layer is responsible for localizing messages for clients of an application. When dealing with exceptions, subclasses of *java.lang.Exception* are recommended for communicating errors. They also need to be serializable so that they can be passed across tiers.

Exception classes should contain information detailing the error and must not already be localized. The presentation tier can use the error information contained in the message to create a message that is appropriate for the client. For the web tier, JSPs are probably the best mechanism for formatting error messages. Uncaught exceptions in a JSP are forwarded to the defined error page, if present. Here is an example:

```
<%@ page language="java" errorPage="jsp/exception/notFound.jsp" %>
```

The *errorPage* element argument defines the URL for the page to be displayed if a JSP page request results in an error.

Message Logging and System Exceptions

Logging messages and system exceptions are intended mostly for support personnel. The recommended approach is to use resource bundles to localize log messages and system exceptions. The easy way to determine a locale for system messages is to use the system default locale. However, if the application happens to be distributed and has differing locales, the locale for all system messages may be defined in an environment entry in the deployment descriptor for the component that creates and outputs the message.

Any system exception should be a subclass of *java.lang.RuntimeException*. With an internationalized design, the exception message should not contain the message itself but a key for the message that can be looked up within a resource bundle. The component that is responsible for writing the message to a log will use the resource bundles and the *MessageFormat* class to build and output a localized exception message. You can create your own subtree below *Exception* and *RemoteException* to indicate exceptions that will return properly localizable error codes.

How Would You Confirm the I18N Status of an Application?

To confirm that I18N validation is successful, you must run the application in another locale where a localized version of it is properly installed and then check that the following statements are true:

- When the application runs in another locale where translated message files are properly installed, the messages and other values that were localized, such as font names and sizes, come from the message files of that locale and not from any hard-coded defaults. This applies to messages from the message class files, startup message files, and install application messages.
- The following information and functional items come from the locale that the application is started in and/or from locale-specific files, as applicable:
  - Images
  - Code templates
  - Text files
  - Font names and sizes
  - Date, time, number, and currency formatting
  - Collation
- Multibyte characters print correctly from the application.

- If a user runs the application in a locale that does not have the application installed correctly, the application falls back to using the messages and other nonmessage localizable resources for the default locale. Localizable nonmessage items include window sizes, font names, and help files.
- Presentation objects resize dynamically to account for both the difference in the size of labels or other information displayed and information that contains multibyte characters.
- Presentation objects that receive keyboard input can receive it from different keyboards, and all legal keys are recognized.
- Browsers, their encoding choices, as well as policies work as expected.
- For multibyte and extended ASCII within files and directory names,
  - Directory names with multibyte as part of the paths to various files used in the application are processed correctly.
  - Multibyte encoding, where legal in files, is parsed and shown correctly.
- Message files are reviewed and deemed clear for translation.
- Any registration, comments, and installation functionality and procedures, including any involving e-mail or web transmissions of information that may include multibyte information, was validated to ensure that the information is processed and received correctly.

CERTIFICATION SUMMARY

An application must be internationalized, and then it can be localized. The Java APIs provide tools that enable the developer to internationalize a JEE application. A JEE application requires the correct behavior for locale and character-set encoding when accepting input, communicating data between tiers, and presenting data back to the client. An application also needs to report system errors in all tiers in a language appropriate for support personnel.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 9.

State Three Aspects of any Application That Might Need to Be Varied or Customized in Different Deployment Locales

- ❑ Presentation of text, dates, numbers
- ❑ Labels on presentation components
- ❑ Sounds
- ❑ Colors
- ❑ Images or icons
- ❑ Input and output routines that read and write text files
- ❑ Collation or ordering of data presented in a list

List Three Features of the Java Programming Language That Can Be Used to Create an Internationalizable/Localizable Application

- ❑ *java.util.Properties* for obtaining localized values using the same key
- ❑ *java.text.NumberFormat* to handle numbers and currencies
- ❑ *java.text.DateFormat* to handle date and time
- ❑ *java.text.Collator* and *java.text.CollationKey* for ordering data
- ❑ *java.text.MessageFormat*, *java.util.ResourceBundle*, or *java.util.PropertyResourceBundle* to handle text
- ❑ *java.io.InputStreamReader* and *java.io.OutputStreamWriter* for reading and writing files
- ❑ *java.util.Locale* and `contentType` and `pageEncoding` attributes for JSPs
- ❑ *java.util.Locale* and `ServletResponse.setContentType()` and `ServletResponse.setLocale()` methods for servlets

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there may be more than one correct answer. Choose all correct answers for each question.

State Three Aspects of Any Application That Might Need to Be Varied or Customized in Different Deployment Locales

1. Why is Internationalization called I18N?
   A. Because the internationalization is just too long to write in a presentation.
   B. The number of characters between the first and last character is 18.
   C. Because I18N is the encoded UTF version of internationalization.
   D. Because I18N is the default ISO code for internationalization.
2. Which of the following application aspects can be customized for different locales?
   A. Labels
   B. Reading a text file
   C. Ordering of data presented in a list
   D. Writing a text file

List Three Features of the Java Programming Language That Can Be Used to Create an Internationalizable/Localizable Application

3. What statement is true with respect to Unicode?
   A. Unicode provides a standard encoding for the character sets of different languages.
   B. Unicode is an encoding that is dependent of the platform, program, or language used to access said data.
   C. Unicode provides a non-unique number for every character.
   D. The Unicode Standard has not been adopted by Microsoft.
4. What statement is true with respect to a *ResourceBundle*?
   A. A *ResourceBundle* allows you to have various lookup tables based on the locale (language/country) upon which the server but not the client is running.
   B. Resource bundles support a fourth-level descriptor beyond language/country variants, such that you can have customized messages for people in the northern and people in the southern part of a country, for example.

C. A *ResourceBundle* is a Hashtable that maps strings to values.

D. Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, it can load the resource from the resource bundle that is appropriate for the current user's locale.

5. When using *ResourceBundle*, what is the procedure the system uses to determine which bundle to bind?

A. The resource bundle lookup searches for classes with various suffixes on the basis of the desired locale and the current default locale as returned by `Locale.getDefault()`, and the root resource bundle (base class), in the following order: from parent level to lower level.

B. The resource bundle lookup searches for classes with various suffixes on the basis of the desired locale and the current default locale as returned by `Locale.getHelp()`, and the root resource bundle (base class), in the following order: from lower level (more specific) to parent level (less specific).

C. The resource bundle lookup searches for classes with various suffixes on the basis of the desired locale and the current default locale as returned by `Locale.getDefault()`, and the root resource bundle (base class), in the following order: from lower level (more specific) to parent level (less specific).

D. The resource bundle lookup searches for classes with various suffixes on the basis of the desired locale and the current default locale as returned by `Locale.getDefault()`,and the root resource bundle as returned by `Locale.getBase()`, in the following order: from parent level (less specific) to lower level (more specific).

6. How do you determine the default character encoding for file operations, JDBC requests, and so on?

A. You can identify the default file encoding by checking the *Jvm* property named *default.properties*, as follows:
```
System.out.println(Jvm.getProperty("default.encoding"))
```

B. The default encoding used by locale/encoding-sensitive API in the Java libraries is determined by the system property *defaultfile.encoding*.

C. You can identify the default file encoding by checking the system property named *file.properties*, as follows:
```
System.out.println(System.getProperty("file.encoding"))
```

D. You can set the default file encoding by checking the *Jvm* property named *default.properties*, as follows:
```
System.setProperty("default.encoding"))
```

7. What does UTF stand for?

A. Universal Technical Frontend

B. Unicode Transformation Format

C. United Text Format
D. Universal Transformation Formula

8. What internationalization areas does Java not support?
A. Locales such as country, regional, or area/cultural identifiers
B. Localized resources by virtue of the *ResourceBundle* series of classes
C. Formatting for dates, numbers and decimals, and messages
D. Planetary variants

9. How can you handle input of different decimal symbols—for example, *343,4* as opposed to *343.4* ?
A. Use *NumberFormat* and its methods `format()` and `parse()`. This will handle the default locale for you.
B. Use *Format* and its methods `format()` and `parse()`. This will handle the default locale for you.
C. Use *DecimalFormat* and its methods `format()` and `parse()`. This will handle the default locale for you.
D. Use *Format* and its methods `numberformat()` and `numberparse()`. This will handle the default locale for you.

10. What is the difference between UTF-8 and UTF-16?
A. UTF-16 represents every character using two bytes. UTF-8 uses the one-byte ASCII character encodings for all languages except English.
B. UTF-16 represents every character using two bytes. UTF-8 uses three bytes per character for all languages except English.
C. UTF-16 represents every character using two bytes. UTF-8 uses the one-byte ASCII character encodings for ASCII characters and represents non-ASCII characters using variable-length encoding.
D. UTF-16 represents every character using one byte. UTF-8 uses the two-byte ASCII character encodings for ASCII characters and uses three bytes per character for all languages except English.

11. What is a locale and how is it used for I18N?
A. A locale is an object that represents and provides information about a specific geographical, political, or cultural region. An operation that requires a locale to perform its task is called locale-sensitive and uses the locale to format information correctly for the user.
B. A locale is an object that represents and provides information about a specific geographical, political, or cultural region. A globale is an object that represents and provides information about a geographical, political, or cultural region.

C. A locale is an object that Java calls to present information to the user based upon the locale location of the browser.

D. A locale is an object that represents the supported geographical, political, or cultural regions. An operation that requires a locale to perform its task is called locale-intensive and uses the locale to display information for the user.

12. Which of the following are logical fonts in Java?

A. Sans-serif

B. Time New Roman

C. Monospaced

D. Dialog

SELF TEST ANSWERS

State Three Aspects of Any Application That Might Need to Be Varied or Customized in Different Deployment Locales

1. ☑ **B** is correct. The number of characters between the first and last character is 18.
   ☒ **A, C,** and **D** are untrue.
2. ☑ **A, B, C** and **D**. All four application aspects can be customized for different locales.

List Three Features of the Java Programming Language That Can Be Used to Create an Internationalizable/Localizable Application

3. ☑ **A** is correct. Unicode provides a standard encoding for the character sets of different languages.
   ☒ **B, C,** and **D** are incorrect. Unicode is an encoding that is independent of the platform, program, or language used to access said data. Unicode provides a unique number for every character. The Unicode Standard has been adopted by Microsoft, as well as by Apple, HP, IBM, JustSystem, Oracle, SAP, Sun, Sybase, Unisys, and others.
4. ☑ **D** is correct. Resource bundles contain locale-specific objects. When your program needs a locale-specific resource, your program can load it from the resource bundle that is appropriate for the current user's locale.
   ☒ **A**, **B,** and **C** are incorrect. A *ResourceBundle* allows you to have various lookup tables based upon what locale (language/country) the client's browser is running in. Resource bundles also support a third-level descriptor beyond language/country, such that you can have customized messages for presentation beyond just language and country. A *ResourceBundle* is analogous to a Hashtable that maps strings to values.
5. ☑ **C** is correct. The resource bundle lookup searches for classes with various suffixes on the basis of the desired locale, the current default locale as returned by `Locale.getDefault()`,and the root resource bundle (base class), in the following order: from lower level (more specific) to parent level (less specific).
   ☒ **A, B,** and **D** are incorrect, as they are at odds with the correct answer, **C**.
6. ☑ **C** is correct. You can identify the default file encoding by checking the *System* property named *file.properties*, as follows:
   ```
   System.out.println(System.getProperty("file.encoding"))
   ```
   ☒ **A, B,** and **D** are incorrect, as they are at odds with the correct answer, **C**.

7. ☑ **B** is correct. UTF stands for Unicode Transformation Format.
☒ **A, C,** and **D** are incorrect.

8. ☑ **D** is correct. Calendar and planetary variants.
☒ **A, B,** and **C** are true. Java internationalization supports locales such as country, regional, or area/cultural identifiers, as well as localized resources by virtue of the *ResourceBundle* series of classes and formatting for dates, numbers and decimals, and messages.

9. ☑ **A** and **C** are correct. Use *NumberFormat* or *DecimalFormat* and its methods `format()` and `parse()`. This will handle the default locale for you.
☒ **B** and **D** are incorrect, as they are at odds with the correct answers, **A** and **C**.

10. ☑ **C** is correct. UTF-16 represents every character using two bytes. UTF-8 uses the one-byte ASCII character encodings for ASCII characters and represents non-ASCII characters using variable-length encoding.
☒ **A, B,** and **D** are incorrect, as they are at odds with the correct answer, **C**. UTF-16 represents every character using two bytes. UTF-8 uses the one-byte ASCII character encodings for ASCII characters and represents non-ASCII characters using variable-length encodings. Note that while UTF-8 can save space for Western languages, which are the most common, it can actually use up to three bytes per character for other languages.

11. ☑ **A** is correct. A locale is an object that represents and provides information about a specific geographical, political, or cultural region. An operation that requires a locale to perform its task is called locale-sensitive and uses the locale to refine and properly format the date and numeric information for the user.
☒ **B, C,** and **D** are incorrect.

12. ☑ **A, C,** and **D** are correct. Java recognizes five font names—Serif, Sans-serif, Monospaced, Dialog, and DialogInput—along with four font styles—plain, bold, italic, and bolditalic. These are physical fonts but are standard names mapped to physical fonts known to be installed by default on the platform. The mapping is handled by *font.properties*. See the lib/fonts subdirectory of the Java JDK.
☒ **B**. Java does not recognize Times New Roman as platform-standard.