WebSphere Application Server Enterprise Services

Internationalization Service

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices”on page 24 .

Contents

The Internationalization Service

In a distributed client-server environment, application processes may run on different machines configured to different locales corresponding to different cultural conventions; they can also be located across geographical boundaries. With the advent of Internet based business computational models, like eCommerce, the possibility is ever increasing that applications will have clients and servers operating in different locales and geographical regions. With respect to internationalization, this heterogeneity introduces new challenges to the task of designing a sound client-server infrastructure.

For example, clients and servers could:

- “Reside in computers having different endian architectures or code sets” on page 2
- “Be located in different locales” on page 2
- “Be located in different time zones” on page 3

The traditional solution for solving locale and time zone mismatch problems is to pass one or more extra parameters on all business methods necessary for conveying either the client's locale or time zone to the server. Though simple, this technique has serious limitations within Enterprise JavaBeans applications:

- It is intrusive, requiring that one or more parameters be added to all bean methods in the call chain to locale-sensitive or time zone-sensitive methods.
- It is inherently error-prone.
- It can be impracticable within applications that do not afford modification, such as a legacy application.

The Internationalization Service offers a solution to these challenges. For more information, see “The Internationalization Service solution” on page 3.

Computers with differing endian architectures or code sets

Clients and servers can reside in computers having different endian architectures: a client could reside in a little-endian CPU, while the server code runs in a big-endian one. As a more complex instance, a client may want to invoke a business method on a server running in a code set different from that of the client.

A client-server infrastructure must define precise endian and code set tracking and conversion rules. As prominent reference examples of client-server architectures, both CORBA and J2EE have addressed the problems of endian and code set mismatches. The language neutral CORBA formalism uses byte order indicator in all marshalled data streams to indicate the byte order of the originating machine; in case of an endian mismatch, the receiving side can perform byte swapping for endian correction. The code set mismatch is addressed by CORBA using a comprehensive framework for code set conversion. J2EE has nearly eliminated the aforementioned problems in a unique way by relying on its Java Virtual Machine (JVM), which encodes all string data in UCS-2 and externalizes everything in big endian format.

The JVM employs a set of platform-specific programs for interfacing with the native platform. These programs perform any necessary code set conversions between UCS-2 and the native code set of a platform.

Computers located in different locales

Client and server processes can execute in geographical locations having different time

zones. For example, a Spanish client may invoke a business method upon an object residing on an American server. Some business methods can be locale sensitive in nature; for example, given a business method that returns a sorted list of strings, the Spanish client will expect that list to be sorted according to the Spanish collating sequence, not in the server's English collating sequence. Since data retrieval and sorting procedures run on the server, the locale of the client must be available in order to perform a legitimate sort.

A similar consideration applies in the instances where the server has to return strings containing date, time or currency, exception messages, and so on, formatted according to the client's cultural expectations. Neither the CORBA nor the J2EE specifications have architecturally addressed the locale mismatch problem, and other options involving extra parameters are not practical or have limitations. For example, requiring an extra parameter could require interface changes, which is a serious concern for deployed applications.

Computers located in different time zones

Client and server processes can execute in geographical locations having different time zones. To date, all internationalization literature and resources have concentrated mainly on code set and locale related issues. They have mostly ignored the time zone issue, even though business methods can be sensitive to time zone as well as to locele.

For example, consider a simple eCommerce application for buying, selling, or trading stocks based on requests originating from its subscribers. Assume that the server is placed in Wall Street in a computer configured for the Eastern Standard Time (EST) time zone. Depending on market volatility, the result of a stock purchase request originating from a Central Standard Time (CST) client can vary dramatically if the eCommerce application does not account for the time zone differential between client and server.

Other time zone sensitive operations include time stamping messages logged to a server and resource (file, database) accesses. The concept of Daylight Savings Time (DST) further complicates the time zone issue. Neither the CORBA nor the J2EE specifications address the issues adequately and traditional methods of solving the problem are limited.

The Internationalization Service solution

The WebSphere Enterprise Internationalization Service solves the locale and time zone mismatch problems without the traditional limitations by managing the distribution of Internationalization context (locale and time zone) across the various components of Enterprise JavaBean applications, including Java client applications, Enterprise JavaBeans, and servlets. Server-side components can use the Internationalization context API to access distributed Internationalization context and then localize computations according to the locale or time zone of remote, client-side components.

The service works by associating an Internationalization context with every thread of execution within an application. When a client-side program invokes a remote business method, the Internationalization Service transparently interposes by obtaining the context associated with the current thread and attaching it to the outgoing request. At the server-side, the Internationalization Service again interposes by detaching the caller’s context from the incoming request and associating it with the thread on which the remote business method will run. The service will propagate this context on subsequent remote business method invocations in the same manner and thus distribute the context of the originating request over the call chain.

This basic operation is precisely defined by the service’s Internationalization context management policies, which specify how context propagates over all supported component types, the context under which a component executes, and the usage guidelines an end-user will follow to programmatically access context using the Internationalization

context API.

To programmatically manage Internationalization context, an application component first resolves the appropriate Internationalization Service API references. Depending on the API reference employed and the applicable context management policy, a component can access different types of Internationalization context elements, such as the caller locale or the invocation time zone.

For information about Internationalization context types and how to programmatically manage them, see:

- “Internationalization context” on page 13
- “Managing Internationalization context” on page 7
- “Internationalization context application programming interface” on page 15

To utilize Internationalization context within an application, the service must be enabled in the Enterprise JavaBeans client container (if the application has an Enterprise JavaBeans client program) and in all application servers containing the application’s Enterprise JavaBeans and servlets.

Information about configuring Internationalization service function appear in:

- “Enabling or disabling the Internationalization Service” on page 5
- “Tracing Internationalization Service Function” on page 11

Using the Internationalization Service

This section discusses the tasks involved in setting up and using the Internationalization Service. Topics included in this section include:

Enabling or disabling the Internationalization Service

This section describes how end-users enable and disable the Internationalization Service to manage context within application servers and within Enterprise JavaBean contained client programs.

For more information, see:

Enabling Internationalization context within application servers

Any Enterprise JavaBean or Servlet can utilize Internationalization context whenever the Internationalization Service is enabled within the containing application server.

The Internationalization Service is enabled within application servers using the **Custom** tab. The **Custom** tab lists all Enterprise Services available to a particular application server and provides access to the configuration properties of each.

Perform the following steps to enable Internationalization context within an application server:

1. View the **Custom** tab:
   a. Start the WebSphere Administration Client tool.
   b. Select the application server requiring the Internationalization Service.
   c. In the right-hand pane, click the **Custom** tab.
      - If the **Custom** tab does not list the Internationalization Service entry, a new entry for Internationalization Service must be added. Go to "step 2" on page 5.
      - If the **Custom** tab lists the Internationalization Service entry, then the Internationalization Service properties can be viewed and edited. Go to "step 3" on page 6.
2. Add a new Internationalization Service entry to the **Custom** tab:
   a. From the **Custom** tab, click **Add**.
   The Add Custom Service dialog is displayed.
   b. In the Add Custom Service dialog, enter the properties as indicated in "step 3" on page 6.
   c. Click **OK**.
   The Add Custom Service dialog closes.

d Click **Apply**.
Your changes are saved.

3. Verify or edit the Internationalization Service properties:
   a From the **Custom** tab, click **Edit**.
   The Edit Custom Service dialog is displayed.
   b Click the **General** tab.
   c In the **Classname** field, verify or type the following entry:
   ```
   com.ibm.ws.i18n.context.ServiceInit
   ```
   d If not already selected, select the **Enabled** check box.
   e Click **OK**.
   The Edit Custom Service dialog closes.
   f Click **Apply** to save any changes.
   Your changes are saved.

These settings cause the Internationalization Service to initialize when starting or restarting the corresponding application server.

Enabling Internationalization context within Enterprise JavaBean Java application clients

The Internationalization Service is enabled to manage Internationalization context within contained Enterprise JavaBean Java application clients whenever the i18nctx.jar file appears in the CLASSPATH constructed by the **launchclient** utility. When invoking a client application, **launchclient** correctly configures the CLASSPATH to include i18nctx.jar, then initializes the service for use within that application.

Disabling Internationalization context within application servers

The Internationalization Service can be disabled within application servers using the **Custom** services tab. The **Custom** services tab lists all Enterprise Services available to a particular application server and provides access to the configuration properties of each.

Perform the following steps to disable Internationalization context within an application server:

1. View the **Custom** services tab:
   a Start the WebSphere Administration Client tool.
   b Select the application server for which you want to disable the Internationalization context.
   c In the right-hand pane, click the **Custom** tab.
   - If the **Custom** tab lists the Internationalization Service entry and indicates that it is enabled, then go to “step 2” on page 6to disable the Internationalization context.
   - Otherwise, the Internationalization Service is disabled and no further action is required. Close the WebSphere Administration Client tool.
2. Edit the Internationalization Service configuration properties:
   a From the **Custom** tab, click **Edit**.
   The Edit Custom Service dialog is displayed.
   b Clear the **Enabled** box.
   c Click **OK**.

The Edit Custom Service dialog closes. Your changes are saved.

These settings keep the Internationalization Service from initializing (that is, they disable the Internationalization context) when starting or restarting the corresponding application server.

Disabling Internationalization context within Enterprise JavaBean Java application clients

The Internationalization Service is normally enabled to manage Internationalization context within contained Enterprise JavaBean Java application clients. To disable Internationalization context, you must ensure that the `i18nctx.jar` Jar file is not included in the CLASSPATH.

Because the CLASSPATH is included automatically when the **launchclient** utility constructs the CLASSPATH, you must remove the `i18nctx.jar` Jar file from the *`WAS_HOME`*`/lib` directory (where *`WAS_HOME`* represents the directory in which WebSphere Advanced Edition is installed). This prevents the file from being included in the CLASSPATH constructed by the **launchclient** utility.

Caution: Removing the `i18nctx.jar` Jar file from the *`WAS_HOME`*`/lib` directory disables any application server in your installation from using the Internationalization Service.

Managing Internationalization context

Enterprise JavaBean components can programmatically manage Internationalization Context using the Internationalization context Application Programming Interface (API). This section describes how to access both caller and invocation Internationalization context elements within Enterprise JavaBean application clients, servlets, and Enterprise JavaBeans.

For additional information, see:

Configuring the programming environment

The `java.util` and `com.ibm.websphere.i18n.context` packages contain all classes necessary to utilize the Internationalization Service within an Enterprise JavaBean application.

Classes specific to the Internationalization Service reside in the *`WAS_HOME`*`/lib/i18nctx.jar` Jar file (where *WAS_HOME* is the directory in which WebSphere Enterprise Extensions is installed). Be sure to add the `i18nctx.jar` Jar file to the CLASSPATH when compiling application components that import Internationalization Service classes.

Gaining access to the Internationalization context API

In the Enterprise JavaBeans environment, the Internationalization Service supplies a JNDI binding to an implementation of the UserInternationalization interface under the java:comp/websphere/UserInternationalization name. Applications requiring access to the service can perform a lookup on that JNDI name, as shown in the following code snippet:

```
import com.ibm.websphere.i18n.context.*;
 import javax.naming.*;
 public class MyApplication {
 ...
 //-------------------------------------------------------------
 // Resolve a reference to the UserInternationalization interface.
 //-------------------------------------------------------------
 InitialContext initCtx = null;
 UserInternationalization userI18n = null;
 try {
   initCtx = new InitialContext();
   userI18n = (UserInternationalization)initCtx.lookup(
   "java:comp/websphere/UserInternationalization");
 }
 catch (NamingException nnfe) {
   // UserInternationalization URL is unavailable.
 }
 catch (NamingException ne) {
   // InitialContext could not be instantiated.
 }
 ...
```

Once an application component resolves a reference to the UserInternationalization interface, it can use this reference to create references to the Internationalization and InvocationInternationalization interfaces, which afford access to both caller and invocation locales and time zone. See the following code snippet:

```
...
 //-------------------------------------------------------------
 // Resolve references to the Internationalization and
 // InvocationInternationalization interfaces.
 //-------------------------------------------------------------
 Internationalization callerI18n = null;
 InvocationInternationalization invocationI18n = null;
 try {
   callerI18n = userI18n.getCallerInternationalization();
   invocationI18n = userI18n.getInvocationInternationalization();
 }
 catch (IllegalStateException iae) {
   // An Internationalization interface(s) is unavailable.
 }
 ...
```

Suggestion: Internationalization context API references need to be resolved only once over the lifecycle of any application component. Therefore, when developing server-side application components (for example, servlets and Enterprise JavaBeans), resolve the Internationalization context API references within the initialization methods of such components (for example, within the `init()` method of servlets, or within the `ejbCreate()` method of Enterprise JavaBeans). See “Programming examples” on page 17 for more information.

Accessing caller locales and time zone

Every invocation of an application component has an associated caller Internationalization context associated with the thread running that invocation. Caller context is propagated using the Internationalization Service and middleware to the target of a request, such as remote Enterprise JavaBean or servlet service methods.

To obtain caller locales and time zone, an application component first resolves a reference to the Internationalization interface, and then calls the appropriate accessor method. Details for obtaining this reference can be found in “Gaining access to the Internationalization context API” on page 7 .

The Internationalization interface contains the following methods to get caller Internationalization context elements:

Locale [] getLocales()

Returns the list of caller locales associated with the current thread.

Locale getLocale()

Returns the first in the list of caller locales associated with the current thread.

TimeZone getTimeZone()

Returns the caller SimpleTimeZone associated with the current thread.

For complete information about Internationalization interface methods, see sections:

- “Internationalization Context Application Programming Interface” on page 15
- “Programming Examples” on page 17

The following code snippet illustrates the basic usage of the Internationalization interface:

```
//-------------------------------------------------------------
// Internationalization context imports.
//-------------------------------------------------------------
import com.ibm.websphere.i18n.context.*;
...
public class MyApplication {
...
//-------------------------------------------------------------
// Resolve the Internationalization context API here. See the
// “Gaining access to the Internationalization Context API” on page 7
// topic for complete details.
//-------------------------------------------------------------
UserInternationalization userI18n   = null;
Internationalization     callerI18n = null;
...
//-------------------------------------------------------------
// Obtain the desired Internationalization context element.
//-------------------------------------------------------------
java.util.Locale [] myLocales       = callerI18n.getLocales();
java.util.Locale myLocale           = callerI18n.getLocale();
java.util.SimpleTimeZone myTimeZone = callerI18n.getTimezone();
...
//-------------------------------------------------------------
// Utilize the caller context element to perform a locale or
// time zone sensitive computation, for example:
//-------------------------------------------------------------
DateFormat df = DateFormat.getDateInstance(myLocale);
String localizedDate = df.getDateInstance().format(aDateInstance);
...
```

Internationalization interface methods are utilized in the same manner and are available within all Enterprise JavaBean application components; however their semantics vary slightly depending upon a component's type. For instance, when obtaining a caller Internationalization context element within a Java client application the service returns the corresponding default or process-based element; in contrast, when obtaining caller locales within a servlet service method (for example, doPost() or doGet()), the service returns the locales propagated within the corresponding HTML request. See “Internationalization context management” on page 14 for a discussion of how the service propagates Internationalization context throughout an application.

Accessing invocation locales and time zone

Every invocation of an application component has an associated invocation Internationalization context associated with the running thread. Invocation context is that under which a request, such as a remote business method implementation, executes; it is propagated on subsequent invocations using the Internationalization Service and middleware.

To access invocation locales and time zone, an application component must first resolve a reference to the InvocationInternationalization interface of the Internationalization context API. For more information, see “Gaining access to the Internationalization context API” on page 7 .

The InvocationInternationalization interface contains methods to both get and set invocation Internationalization context elements:

Locale [] getLocales()

Returns the list of invocation locales associated with the current thread.

Locale getLocale()

Returns the first in the list of invocation locales associated with the current thread.

TimeZone getTimeZone()

Returns the invocation SimpleTimeZone associated with the current thread.

setLocales(Locale [])

Sets the list of invocation locales associated with the current thread to the supplied list.

setLocale(Locale)

Sets the list of invocation locales associated with the current thread to a list containing the supplied locale.

setTimeZone(TimeZone)

Sets the invocation time zone associated with the current thread to the supplied SimpleTimeZone.

setTimeZone(String)

Sets invocation time zone associated with the current thread to a SimpleTimeZone having the supplied ID.

For complete information about Internationalization interface methods, see:

- “Internationalization Context Application Programming Interface” on page 15
- “Programming Examples” on page 17

The following code snippets illustrate the use of the InvocationInternationalization interface:

```
//-------------------------------------------------------------
 // Internationalization context imports.
 //-------------------------------------------------------------
 import com.ibm.websphere.i18n.context.*;
 ...
 public class MyApplication {
 ...
 //-------------------------------------------------------------
 // Resolve the Internationalization context API. For details,
 // see “Gaining access to the Internationalization Context API” on
page 7
 .
 //-------------------------------------------------------------
 UserInternationalization userI18n = null;
 Internationalization invocationI18n = null;
 ...
 //-------------------------------------------------------------
 // Obtain the desired invocation context element.
 //-------------------------------------------------------------
 java.util.Locale []      myLocales  = invocationI18n.getLocales();
 java.util.Locale         myLocale   = invocationI18n.getLocale();
 java.util.SimpleTimeZone myTimeZone = invocationI18n.getTimezone();
 ...
 //-------------------------------------------------------------
 // Utilize an invocation context element to perform a locale or
 // time zone sensitive computation, for example:
 //-------------------------------------------------------------
 DateFormat df = DateFormat.getDateInstance(myLocale);
 String localizedDate = df.getDateInstance().format(aDateInstance);
 ...
```

The InvocationInternalization interface allows read and write access to invocation Internationalization context within application components. However, according to

Internationalization context management policies, only Enterprise JavaBean application clients have write access to invocation Internationalization context elements. Differences in how application components may utilize InvocationInternationalization methods are explained in “Internationalization context management” on page 14

In the following code snippet, locale (en,GB) and simple time zone (GMT) transparently propagate on the call to myBusinessMethod(). Server-side application components, such as myEjb, may utilize the InvocationInternationalization interface to obtain these context elements.

```
...
 //-------------------------------------------------------------
 // Set the invocation context that will propagate on subsequent
 // remote business method calls.
 //-------------------------------------------------------------
 Locale localeToPropagate = new Locale("en", "GB");
 SimpleTimeZone timeZoneToPropagate =
     (SimpleTimeZone)SimpleTimeZone.getTimeZone("GMT");
 invocationI18n.setLocale(localeToPropagate);
 invocationI18n.setTimeZone(timeZoneToPropagate);
 myEjb.myBusinessMethod();
```

Internationalization context management policies also stipulate that server-side application components (for example, Enterprise JavaBeans and servlets) always run under the caller's context, if it exists, or otherwise that of the containing server process. In addition, such components cannot set invocation context elements.

Thus, within server-side application components the Internationalization and InvocationInternationalization interfaces are semantically equivalent, and either can be used to obtain the context associated with the thread on which that component is running. For instance, both interfaces can be used to obtain the list of locales propagated to a servlet service `doPost()`.

Because the model specifies read-only access to server-side invocation context, calls to set invocation context elements within server-side application components result in a `java.lang.IllegalStateException`.

Tracing Internationalization Service function

You can enable an application server to emit trace statements regarding Internationalization Service function by specifying the trace string to an application server's Trace Service.

To enable trace using an application server's Trace Service:

1. In the WebSphere Advanced Administrative Console window, expand **WebSphere administrative domain > Nodes > *node* > Application servers > *application_server*** .

   The properties related to the selected application server are displayed in the right-hand pane.

   Note: *node* represents the node on which the application server is located and *application_server* represents the name of the application server for which you are enabling the trace.

2. In the right-hand pane of the WebSphere Advanced Administrative Console window, select the **Services** tab.
3. On the **Services** tab, select **Trace services** from the list of services.
4. On the **Trace services** pop-up menu, click **Edit Properties**.

   The Trace Service dialog is displayed.

5. Click **Edit Properties**.

   The Trace Service Properties dialog is displayed.

6. In the **Trace Specification** box, enter the following as a continuous string (no spaces and no line breaks):

```
com.ibm.ws.i18n.context.*=all=enabled:
   com.ibm.websphere.i18n.context.*=all=enabled:
   com.ibm.II18nContextImpl.*=all=enabled:
   com.ibm.II18nContextImpl.util.*=all=enabled
```

7. Click **OK**.

   The Trace Service Properties dialog closes.

8. Click Apply.

   The changes are applied.

   Notes:

1. Clicking **Apply** enables the trace and makes the trace persistent across server shutdown and restart.
2. Within this procedure, you can specify an output file to which to log application server trace messages, including those generated by the Internationalization Service.

Restart the application server. Messages tracing function of the Internationalization Service output to the file specified in the Trace Service Properties dialog.

Examples and reference

This section provides the reference information you can use to assist you when setting up and using the Internationalization Service. Topics included in this section include:

The programming model

This section describes the types used to programmatically compose and manage Internationalization context within an Enterprise JavaBean application. All programming types mentioned in this section are contained in the java.util and com.ibm.websphere.i18n.context packages.

For more information, see:

Internationalization context

Internationalization context consists of a fixed-length list (array) of locales and a time zone (where a locale is an instance of the java.util.Locale class and time zone is an instance of the java.util.SimpleTimeZone class). Refer to the Java SDK API documentation for a complete description of each type.

Note: For this release, the Internationalization Service does not support Java SDK TimeZone types other than java.util.SimpleTimeZone. Unsupported TimeZone types silently map to default SimpleTimeZone when supplied to service API methods.

Enterprise JavaBean applications use the service to access and manage the following types of Internationalization context:

Caller context

Caller context is a locale list and time zone that propagates from calling application components on remote business methods and requests; it is the context associated with an incoming request. Caller context is accessible within all application components, but is manageable neither declaratively nor programmatically and defaults to that of the process (for example, `java.util.Locale.getDefault()` or `java.util.SimpleTimeZone.getDefault()`) whenever it is unavailable.

Invocation context

Invocation context is a locale list and time zone that propagates to target application components on remote business methods; it is the context under which a component or business method executes. Invocation context is manageable both declaratively and programmatically according to the applicable Internationalization Service context management policies. In particular, these polices specify API access restrictions, how invocation context propagates on remote requests, and the context under which a target request executes.

To make these terms more concrete, imagine a simple JavaBean application having a client

that invokes remote bean method, `myBeanMethod()`. On the client-side, the application can utilize the service to access caller or invocation context. When the application calls `myBeanMethod()`, the service attaches the invocation context to the outgoing request. On the server-side, the service detaches the caller context from the incoming request and makes it available to the implementation of `myBeanMethod()`, which can utilize the Internationalization service API to access it.

These terms are important in describing how the WebSphere Application Server manages Internationalization context on behalf of an application.

Internationalization context management

The Internationalization Service transparently propagates locales and time zone across the various components of an Enterprise JavaBean application, including (contained) Java clients, Enterprise JavaBeans and servlets. How the service propagates Internationalization context is determined by its context management policies.

For this release, the following context management policies correspond to whether an application component is contained on the client-side or on the server-side:

Client-side Internationalization

Enterprise JavaBean client application components deployed within J2EE client containers, only, are subject to the Client-side Internationalization (CSI) policy.

In CSI, client applications can programmatically get and set the invocation context elements (that is, locales and time zone) associated with the current thread by using the Internationalization context API. Invocation context elements set using the API persist until set again or until the application exits. Clients may also get the caller context elements associated with the current thread. In a client application, caller context is always the default locale and time zone of the process.

On remote bean method calls originating from a client application, the service propagates all invocation context elements associated to the current thread with the outgoing request. If an invocation context element is non-null (that is, it was set using the API) the service propagates the specified element; otherwise, the service propagates the default element that is associated with the process at that time.

Server-side Internationalization

Enterprise JavaBean application components deployed within J2EE server-side containers are subject to the Server-side Internationalization (SSI) policy. SSI is a simple, immutable policy that uniformly specifies how Internationalization context propagates across the Web and Enterprise JavaBean container boundaries and defines access privileges to Internationalization context within contained application components.

In SSI, server-side application components always run under the caller's Internationalization context. More specifically, the service associates the context of an incoming business method request to the thread on which that method will execute. The incoming (or caller) context is associated as both the caller and invocation context for that method and persists until the method returns.

This policy is applied directly to Enterprise JavaBean methods: On incoming remote bean method requests, the service associates the caller's context to the thread on which the request will execute. SSI is defined slightly differently with respect to servlet service methods (for example, `doGet()` or `doPost()`) because they are invoked using HTML browser clients: On incoming servlet

service requests originating from an HTML browser client, the service associates the list of locales, propagated by the browser within the HTTP header, and the server's default time zone to the thread on which the servlet service method will execute.

For remote bean method calls originating from either a servlet or Enterprise JavaBean method, the service propagates all invocation context elements associated to the current thread on the outgoing request. If an invocation context element is non-null (that is, it was set using the API) the service propagates the specified element; otherwise, the service propagates the default element associated with the server process at that time. This is identical to CSI, except that in SSI, invocation context is always that of the caller. This management policy could be referred to as *'RunAsCaller/DefaultToServer'*.

Enterprise JavaBeans implementations and Servlets can use the Internationalization service API to programmatically get both caller and invocation context elements (that is, locales and time zone) associated with the current thread, but cannot set them. This restriction enforces the *'RunAsCaller/DefaultToServer'* context management strategy and ensures that the invocation context of every client application request, including those from HTTP browsers, distributes over the span of that request.

Thread association considerations

Internationalization context (locales and time zone) is thread scoped. Methods of the Internationalization context API either associate context to or obtain context associated with the thread on which they execute. In cases where new threads are spawned within an application component (for instance, a user generated thread inside the `service()` method of a servlet or a system generated event handling thread in an AWT client) the Internationalization context associated with the parent thread does not automatically transfer to the newly spawned thread. In such instances, the Internationalization Service propagates the default locale and time zone on any remote method invocations executed on the new thread. If the default context is inappropriate, the desired invocation context elements must be explicity associated to the new thread using the `setXxx()` methods of the InvocationInternationalization interface. Currently, Internationalization context management policies allow invocation context to be set within Enterprise JavaBean client programs, but not within servlets and Enterprise JavaBeans. For additional information, see "The InvocationInternationalization interface" on page 16

The JNDI environment

When the Internationalization Service is enabled, the UserInternationalization interface is available to all application component environments using a JNDI lookup on the Initial Context for URL `java:comp/websphere/UserInternationalization`. If the UserInternationalization interface is unavailable due to an Internationalization Service anomaly or restriction, the JNDI lookup invocation throws a `javax.naming.NamingException` containing the `java.lang.IllegalStateException` exception.

Internationalization context application programming interface

Application components programmatically manage Internationalization context through the UserInternationalization, Internationalization, and InvocationInternationalization interfaces within the com.ibm.websphere.i18n.context package. The following code snippet introduces the Internationalization context application programming interface (API):

```
public interface "UserInternationalization" on page 16
  {
   public Internationalization getCallerInternationalization();
   public InvocationInternationalization
getInvocationInternationalization();
 }
 public interface "Internationalization" on page 16
```

```
  {
   public java.util.Locale[] getLocales();
   public java.util.Locale getLocale();
   public java.util.TimeZonegetTimeZone();
}
public interface “InvocationInternationalization” on page 16
    extends Internationalization {
   public void setLocales(java.util.Locale [] locales);
   public void setLocale(java.util.Locale jmLocale);
   public void setTimeZone(java.util.TimeZonetimeZone);
   public void setTimeZone(String timeZoneId);
}
```

The **UserInternationalization interface** provides factory methods for the Internationalization Service interfaces affording access to the caller and invocation contexts. The UserInternationalization interface declares the following methods:

Internationalization getCallerInternationalization()

Returns a reference implementing the Internationalization interface. If the service is disabled, the method throws an IllegalStateException.

InvocationInternationalization getInvocationInternationalization()

Returns a reference implementing the InvocationInternationalization interface. If the service is disabled, the method throws an IllegalStateException.

The **Internationalization interface** declares methods affording read-only access to caller Internationalization context:

Locale[] getLocales()

Returns the caller locale list associated with the current thread, provided the locale list is non-null; otherwise the method returns a locale list of `length(1)` containing the process locale.

Locale getLocale()

Returns the tail of the caller locale list associated with the current thread, provided the locale list is non-null; otherwise the method returns the process locale.

TimeZone getTimeZone()

Returns the caller time zone (that is, the SimpleTimeZone) associated with the current thread, provided the time zone is non-null; otherwise the method returns the process time zone.

The **InvocationInternationalization interface** declares methods affording read and write access to invocation Internationalization context.

Note: According to the server-side Internationaliztion context management policy, all set methods (*setXxx()*) throw an `IllegalStateException` when called within a server-side application component, such as a servlet or Enterprise JavaBean implementation.

void setLocales(java.util.Locale[] locales)

Sets the invocation locale list element to the supplied locale list (`locales`), within the Internationalization context associated with the current thread. The supplied locale list can be null or have `length(>= 0)`. When the supplied locale list is null or has `length(0)`, the service sets the invocation locale list to an array of `length(1)` containing the default locale. Null entries can exist within the supplied locale list, but the service substitutes the default locale for null on remote invocations.

Locale[] getLocales()

Returns the invocation locale list element of the Internationalization context associated with the current thread, provided the locale list is non-null; otherwise the method returns a locale list of `length(1)` containing the default locale.

void setLocale(java.util.Locale locale)

Sets the invocation locale list element to an array of `length(1)` containing the supplied locale (`locale`), within the Internationalization context associated with the current thread. The supplied locale can be null, in which case the service instead sets the invocation locale list to an array of `length(1)` containing the default locale.

Locale getLocale()

Returns the tail of the invocation locale list element of the Internationalization context associated with the current thread, provided the locale list is non-null; otherwise the method returns the default locale.

void setTimeZone(java.util.TimeZone timeZone)

Sets the invocation time zone element to the supplied time zone (`timeZone`), within the Internationalization context associated with the current thread. If the supplied time zone is not an exact instance of java.util.SimpleTimeZone or is null, the service instead sets the invocation time zone to the default time zone.

void setTimeZone(String timeZoneId)

Sets the invocation time zone element to a java.util.SimpleTimeZone having the supplied ID (`timeZoneId`) within the Internationalization context associated with the current thread. If the supplied time zone ID is null or unsupported (that is, it does not appear in the list of IDs returned by the `java.util.TimeZone.getAvailableIds()` method) the service sets the invocation time zone to a time zone to the simple time zone having an ID of GMT and otherwise invalid fields.

TimeZone getTimeZone()

Returns the invocation time zone element of the Internationalization context associated with the current thread, provided the time zone is non-null; otherwise the method returns the default time zone (java.util.SimpleTimeZone).

Handling exceptions

The Internationalization Service employs one exception:

```
java.lang.IllegalStateException
```

This exception is employed to indicate that an application component attempted an operation not supported by the programming model or that an anomaly occurred that caused the service to disable.

With respect to the programming model, `IllegalStateException` is thrown whenever a server-side application component attempts to set invocation context. This is a violation of the server-side Internationalization (SSI) context management policy. Under SSI, servlets and Enterprise JavaBeans cannot modify their invocation Internationalization context.

The service also throws `IllegalStateException` to indicate the service is disabled. For instance, the JNDI lookup on the UserInternationalization URL throws a `javax.naming.NamingException` containing an instance of IllegalStateException if the service did not properly initialize. In such cases refer to the trace log to determine the reason for failure and if necessary, call IBM Technical Support.

Programming examples

This section illustrates usage of the Internationalization context API within various Enterprise JavaBean application components.

To use the Internationalization Service, all application components must import types supplied in the com.ibm.websphere.i18n.context package. A component gains access to the service by performing a JNDI lookup on the initial context for URL `java:comp/websphere/UserInternationalization` to resolve a reference to the UserInternationalization interface. It can then use this reference to obtain references to either the Internationalization or InvocationInternationalization interfaces, depending on whether the component requires access to caller, or to invocation Internationalization context, respectively. These steps are collectively referred to as *'resolving the Internationalization context API'* within the examples.

Differences in and suggestions for service utilization among the various application components are discussed in comments preceding relevant statement blocks.

For details, see the following examples:

- "Enterprise JavaBeans Java client (contained)" on page 18
- "Enterprise JavaBeans servlet" on page 19
- "Enterprise JavaBeans session bean" on page 21

Enterprise JavaBeans Java client (contained)

The code sample following illustrates how to utilize the Internationalization context API within a contained Enterprise JavaBeans Java client program.

```
//-------------------------------------------------------------------
 // Basic Example: J2EE EJB Client.
 //-------------------------------------------------------------------
 package examples.basic;
 import java.util.Properties;
 import javax.naming.InitialContext;
 import javax.naming.NamingException;
 //-------------------------------------------------------------------
 // INTERNATIONALIZATION SERVICE: Imports.
 //-------------------------------------------------------------------
 import com.ibm.websphere.i18n.context.UserInternationalization;
 import com.ibm.websphere.i18n.context.Internationalization;
 import com.ibm.websphere.i18n.context.InvocationInternationalization;
 import java.util.Locale;
 import java.util.SimpleTimeZone;
 public class EjbClient {
   public static void main(String args[]) {
     InitialContext initCtx = null;
 //-------------------------------------------------------------------
 // INTERNATIONALIZATION SERVICE: API references.
 //-------------------------------------------------------------------
 UserInternationalization       userI18n       = null;
 Internationalization           callerI18n     = null;
 InvocationInternationalization invocationI18n = null;
 // Obtain a reference to the JNDI initial context.
 try {
  initCtx= new InitialContext();
     } catch (Exception e) {
 // Error resolving the JNDI initial context.
     }
 ...
 //-------------------------------------------------------------------
 // INTERNATIONALIZATION SERVICE: Resolve API.
 //
 // Gain access to the Internationalization context API by resolving
 // a reference to the UserInternationalization interface.
 // UserInternationalization is a factory for other interfaces that
 // provide access to different Internationalization context types.
 //
 // Next, obtain references to the Internationalization and/or
 // InvocationInternationalization interfaces. Interface
 // Internationalization provides read access to context elements.
 // Call UserInternationalization method
 // getCallerInternationalization() to obtain a reference for getting
 // CALLER context within all EJB application components.
 // Interface InvocationInternationalization provides read/write
 // access to context elements. Call UserInternationalization method
 // getInvocationInternationalization() to obtain a reference for
 // getting/setting invocation context within EJB Clients, and for
```

```
// getting invocation context within Servlets and EJB
// implementations.
//-------------------------------------------------------------------
final String UserI18nUrl =
  "java:comp/websphere/UserInternationalization";
try {
  userI18n = (UserInternationalization)initCtx.lookup(UserI18nUrl);
  callerI18n = userI18n.getCallerInternationalization();
  invocationI18n = userI18n.getInvocationInternationalization();
    } catch (NamingException e) {
  // Error resolving the UserInternationalization object.
    } catch (IllegalStateException e) {
  // Error creating an Internationalization interface reference.
    }
    ...
//-------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Set invocation locale and time zone.
//
// Under the Client-side Internationalization (CSI) context management
// policy, contained EJB client programs may set invocation context
// elements. The following statements associate the
// supplied invocation locale and time zone with the current thread.
// Subsequent remote bean method calls will propagate these context
// elements.
//-------------------------------------------------------------------
invocationI18n.setLocale(new Locale("fr", "FR", ""));
invocationI18n.setTimeZone("ECT");
...
//-------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Get locale and time zone.
//
// Under CSI, contained EJB client programs may get caller and
// invocation Internationaliztion context elements associated with
// the current thread. The next four statements return the invocation
// locale and time zone associated above, and the caller locale and
// time zone associated internally by the service. Getting a caller
// context element within a contained client results in the default
// element.
//-------------------------------------------------------------------
Locale         invocationLocale   =
                      invocationI18n.getLocale();
SimpleTimeZone invocationTimeZone =
                      (SimpleTimeZone)invocationI18n.getTimeZone();
Locale         callerLocale       =
                      callerI18n.getLocale();
SimpleTimeZone callerTimeZone     =
                      (SimpleTimeZone)callerI18n.getTimeZone();
...
  } // main
  ...
} // EjbClient
```

Enterprise JavaBeans servlet

The example following suggests how to utilize the Internationalization context application programming interface (API) within a servlet. Note the `init()` and `doPost()` methods.

```
...
//-------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Imports.
//-------------------------------------------------------------------
import com.ibm.websphere.i18n.context.*;
public class J2eeServlet extends HttpServlet {
  ...
//-------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: API references.
//-------------------------------------------------------------------
UserInternationalization userI18n   = null;
Internationalization     callerI18n = null;
final String UserI18nUrl =
                "java:comp/websphere/UserInternationalization";
...
/**
 * Initialize this Servlet.
 *  <p>Resolve references to the JNDI initial context and the
 * Internationalization context API.
 */
public void init() throws ServletException {
// Resolve the JNDI initial context.
try {
  // JNDI requires the name of the naming context provider and
  // the name of initial context factory. We store these names
  // in a properities file.
  Properties properties = new Properties();
```

```
  properties.put("java.naming.provider.url", "iiop:///");
  initialContext = new InitialContext(properties);
} catch (NamingException ne) {
  throw new ServletException("Cannot resolve JNDI
                              initial conext:" + ne);
}
...
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Resolve API.
//
// Under the Server-side Internationalization (SSI) context
// management policy, Servlets have read-only access to invocation
// context elements. Attempts to set these elements result in an
// IllegalStateException. And because Servlets “RunAsCaller”
// under SSI, the invocation and caller contexts are identical.
// So, this example resolves only a reference to the
// Internationalization interface to get caller locale and
// time zone.
//
// Suggestion: cache all Internationalization context API
// references once, during initialization, and use them
// throughout the Servet lifecycle.
//---------------------------------------------------------------------
try {
  userI18n =
    (UserInternationalization)initialContext.lookup(UserI18nUrl);
  callerI18n = userI18n.getCallerInternationalization ();
} catch (NamingException e) {
  throw new ServletException("Cannot resolve
                              UserInternationalization" + e);
} catch (IllegalStateException e) {
  throw new ServletException("Cannot resolve
                              CallerInternationalization" + e);
}
...
} // init
...
/**
 * Process incoming HTTP GET requests.
 * @param request Object that encapsulates the request to the servlet
 * @param response Object that encapsulates the response from the
 * Servlet.
 */
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
  throws ServletException, IOException {
  doPost(request, response);
} // doGet
/**
 * Process incoming HTTP POST requests
 * @param request Object that encapsulates the request to the
 * Servlet.
 * @param response Object that encapsulates the response from
 * the Servlet.
 */
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
  throws ServletException, IOException {
  ...
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Get HTTP Browser Client Locales.
//
// The Internationalization service associates the locale list
// propagated in the HTTP request header with the current thread.
// It is accessible within HTTP Servlet service and subsequent
// methods via the Internationalization context API.
//
// If the incoming HTTP request does not contain a locale list,
// the Internationalization service associates the server’s default
// locale with the current thread. The service also associates the
// server’s default time zone.
//
// All context elements associate with the current thread will
// propagate over subsequent remote bean methods calls.
//---------------------------------------------------------------------
iLocale = callerI18n.getLocale();
// <WORKAROUND> Browsers are inconsistent regarding how they
// propagate locale information in that they may send locales
// containing a language code, but lacking a country code -
// like ("fr", "") for "French as spoken in France." The
/// following code supplies a default country code in these cases.
if (iLocale.getCountry().equals(""))
  iLocale = customerLocale.getCompleteLocale(iLocale);
// </WORKAROUND>
```

```
// Use iLocale in JDK locale-sensitive operations, etc.
...
  } // doPost
...
} // CLASS J2eeServlet
```

Enterprise JavaBeans session bean

The code snippet following suggests how to perform a localized operation using the Internationalization Service within an Enterprise JavaBeans session bean. Note the comments within the `ejbCreate()` and `getExchangeRate()` methods.

```
...
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: API references.
//---------------------------------------------------------------------
UserInternationalization       userI18n = null;
InvocationInternationalization invocationI18n = null;
final String UserI18nUrl =
                   "java:comp/websphere/UserInternationalization";
...
/**
 * EJB Activate method.
 * @exception java.rmi.RemoteException <EJB Method Requirement>
 */
public void ejbActivate() throws java.rmi.RemoteException {}
/**
 * EJB Create method
 *  <p>Resolve the Internationalization context API.
 * @exception javax.ejb.CreateException whenever this bean cannot
 *  be instantiated.
 * @exception java.rmi.RemoteException <EJB Method Requirement>
 */
public void ejbCreate()
  throws javax.ejb.CreateException, java.rmi.RemoteException {
  // Resolve the JNDI initial context.
  try {
    Properties properties = new Properties();
    properties.put("java.naming.provider.url", "iiop:///");
    initialContext = new InitialContext(properties);
  } catch(NamingException ne) {
    System.out.println("Cannot instantiate JNDI
                        initial context: " + ne);
  }
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Resolve API.
//
// Under the Server-side Internationalization (SSI) context
// management policy, EJBs have read-only access to invocation
// context elements. Attempts to set these elements result in an
// IllegalStateException. And because EJBs “RunAsCaller”
// under SSI, the invocation and caller contexts are identical.
// This example resolves only a reference to the
// InvocationInternationalization interface for getting caller
// context elements.
//
// Suggestion: cache all Internationalization context API
// references once, during instantiation, and use them
// throughout the EJB lifecycle.
//---------------------------------------------------------------------
  try {
    userI18n =
        (UserInternationalization)initialContext.lookup(UserI18nUrl);
    invocationI18n = userI18n.getInvocationInternationalization();
  } catch (NamingException ne) {
    System.out.println ("Cannot resolve
                         UserInternationalization: " + nnfe);
  } catch (IIlegalStateException ise) {
    System.out.println ("Cannot resolve
                         InvocationInternationalization: " + ise);
  }
}
/**
 * ejbPassivate method.
 * @exception java.rmi.RemoteException <EJB Method Requirement>
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbRemove method.
 * @exception java.rmi.RemoteException <EJB Method Requirement>
 */
public void ejbRemove() throws java.rmi.RemoteException {}
/**
```

```
 * getSessionContext method.
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
  return mySessionCtx;
}
/**
 * setSessionContext method.
 * @param ctx the supplied EJB session context.
 * @exception java.rmi.RemoteException <EJB Method Requirement>.
 */
public void setSessionContext(javax.ejb.SessionContext ctx)
  throws java.rmi.RemoteException {
  mySessionCtx = ctx;
}
...
/**
 * Business method.
 * @return the exchange rate associated with the invocation locale
 *  under which this method is executing.
 * @exception java.rmi.RemoteException <EJB Method Requirement>
 */
public getExchangeRate() throws java.rmi.RemoteException {
  ...
//---------------------------------------------------------------------
// INTERNATIONALIZATION SERVICE: Get caller/invocation context.
//
// The Internationalization service associates the locale list
// propagated in the incoming remote bean method request with the
// current thread. It is accessible within the bean method
// implementation and within subsequent methods via the
// Internationalization context API.
//
// If the incoming HTTP request does not contain Internationalization
// context, the service associates the server's default locale
// and time zone with the current thread.
//
// All context elements associated with the current thread will
// propagate over subsequent remote bean methods calls.
//---------------------------------------------------------------------
Locale         iLocale   = invocationI18n.getLocale();
SimpleTimeZone iTimeZone =
                   (SimpleTimeZone)invocationI18n.getTimeZone();
...
// Perform a locale-sensitive computation.
float exchangeRate = exchangeRates.get(iLocale);
...
  return exchangeRate;
} // getRate
...
} // CLASS J2eeSessionBean
```

Verifying service configuration

The Internationalization Service can successfully initialize during an application server's startup process provided the following conditions exist. These conditions apply to both contained Enterprise JavaBean Java client programs and application servers requiring use of the Internationalization service.

- The `i18nctx.jar` file resides in the *`WAS_HOME`*`/lib` subdirectory (where *WAS_HOME* represents the directory in which the WebSphere Application Server product was installed). The `i18nctx.jar` archive file contains all classes and resources necessary to utilize the Internationalization Service.
- The `eex.jar` file resides in the *`WAS_HOME`*`/lib` subdirectory (where *WAS_HOME* represents the directory in which the WebSphere Application Server product was installed) and contains a version of the `eex.xml` file supplying no entries regarding Internationalization Service enablement. The `eex.jar` archive file contains the EE Services initialization framework and a version of the `eex.xml` file containing no Internationalization Service-related entries (see eexr.jar, in the following list item).
- The `eexr.jar` file resides in the *`WAS_HOME`*`/lib/ext` subdirectory (where *WAS_HOME* represents the directory in which the WebSphere Application Server product was installed) and contains a version of the eex.xml file supplying entries

necessary to enable the Internationalization Service.

The `eexr.jar` archive file contains the eex.xml file, which directs the Enterprise Services initialization framework towards classes required to initialize the Internationalization Service. The following code snippet includes XML definitions that are necessary to enable the Internationalization Service on an application server:

```
<?xml version="1.0" ?>
<!DOCTYPE enterprise-extension (View Source for full doctype...)>
- <enterprise-extension>
- <Plugins>
    <class>com.ibm.ws.i18n.context.ServiceInit</class>
  </Plugins>
  <BeforeActivationCollaboratorFactories>
    <class>com.ibm.ws.i18n.context.ServiceInit</class>
  </BeforeActivationCollaboratorFactories>
- <ContextPlugins>
    <class>com.ibm.ws.i18n.context.ServiceInit</class>
  </ContextPlugins>
  <ServletInvocationListenerFactories>
  <class>com.ibm.ws.i18n.context.ServiceInit</class>
  </ServletInvocationListenerFactories>
```

- The `EEservices.xml` file resides in the *WAS_HOME*`/Enterprise/bin` subdirectory (where *WAS_HOME* represents the directory in which the WebSphere Application Server product was installed) and supplies the default entries to enable the Internationalization Service within an application server's Custom service panel.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing IBM Corporation North Castle DriveArmonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:
IBM World Trade Asia Corporation Licensing 2-31Roppongi 3-chome, Minato-ku Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation Department LZKS 11400 Burnet Road Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it

are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

Advanced Peer-to-Peer Networking AFS AIX APPN AS/400 CICS CICS OS/2 CICS/400 CICS/6000 CICS/ESA CICS/MVS CICS/VSE CICSPlex DB2 DB2 Universal Database DCE Encina Lightweight Client DFS Encina IBM IBM System Application Architecture IMS IMS/ESA Language Environment	***	MQSeries MVS/ESA NetView Open Class OS/2 OS/390 OS/400 Parallel Sysplex PowerPC RACF RAMAO RMF RISC System/6000 RS/6000 S/390 SAA SecureWay TeamConnection Transarc TXSeries VSE/ESA VTAM VisualAge WebSphere

	This software contains RSA encryption code.