Preference Object-based Internationalization for Distributed Application Framework in Java

Kenya Ishimoto
kenya@jp.ibm.com
Yamato Software Laboratory
IBM Japan, Ltd.

18th International Unicode Conference

Hong Kong, April 2001

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
- Conclusion
- Q&A

Introduction

Tivoli

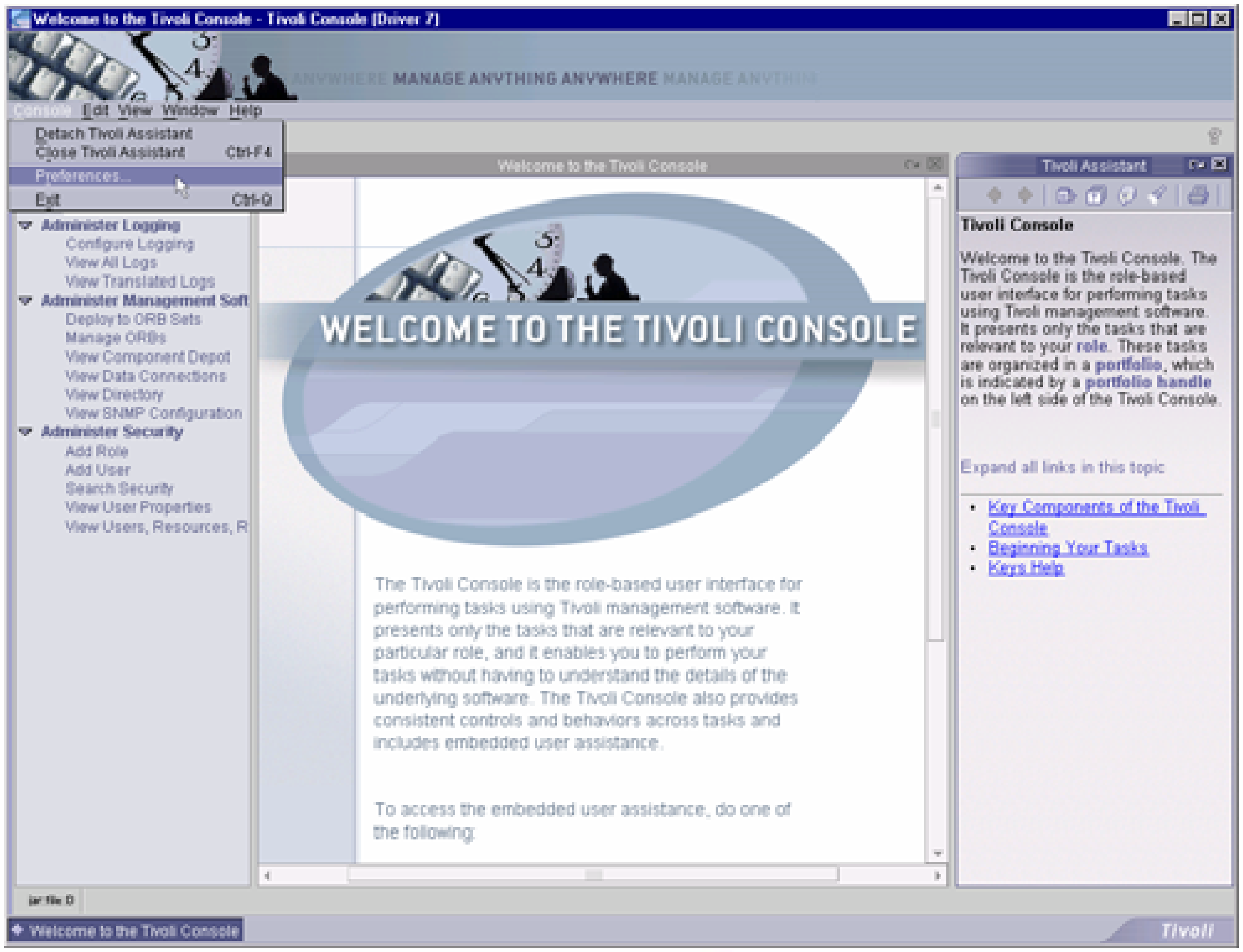
Tivoli Systems - background information

- an IBM company
- dedicated to providing products, services, and programs that enable companies of any size to manage their networked PCs and distributed systems from a single location.

Tivoli Application Framework

- Distributed
- Used in the global company
  - multi-lingual
  - multi-culture
- Multiple-platform
- CORBA, XML
- Java
- UTF-8
- Component-base
- Tivoli Console for applications to interact users

Tivoli Console Example



The Tivoli Console is the role-based user interface for performing tasks using Tivoli management software. It presents only the tasks that are relevant to your particular role, and it enables you to perform your tasks without having to understand the details of the underlying software. The Tivoli Console also provides consistent controls and behaviors across tasks and includes embedded user assistance.

To access the embedded user assistance, do one of the following:

Tivoli Console

Welcome to the Tivoli Console. The Tivoli Console is the role-based user interface for performing tasks using Tivoli management software. It presents only the tasks that are relevant to your **role**. These tasks are organized in a **portfolio**, which is indicated by a **portfolio handle** on the left side of the Tivoli Console.

Tivoli

Agenda

- Introduction
- Requirements for Internationalization
  1. Global multi-user support
  2. Multiple locales for a single user
  3. End-user Customization
  4. Advanced Features
- Difficulties with JDK
- Approach
- Conclusion
- Q&A

Requirements for Internationalization

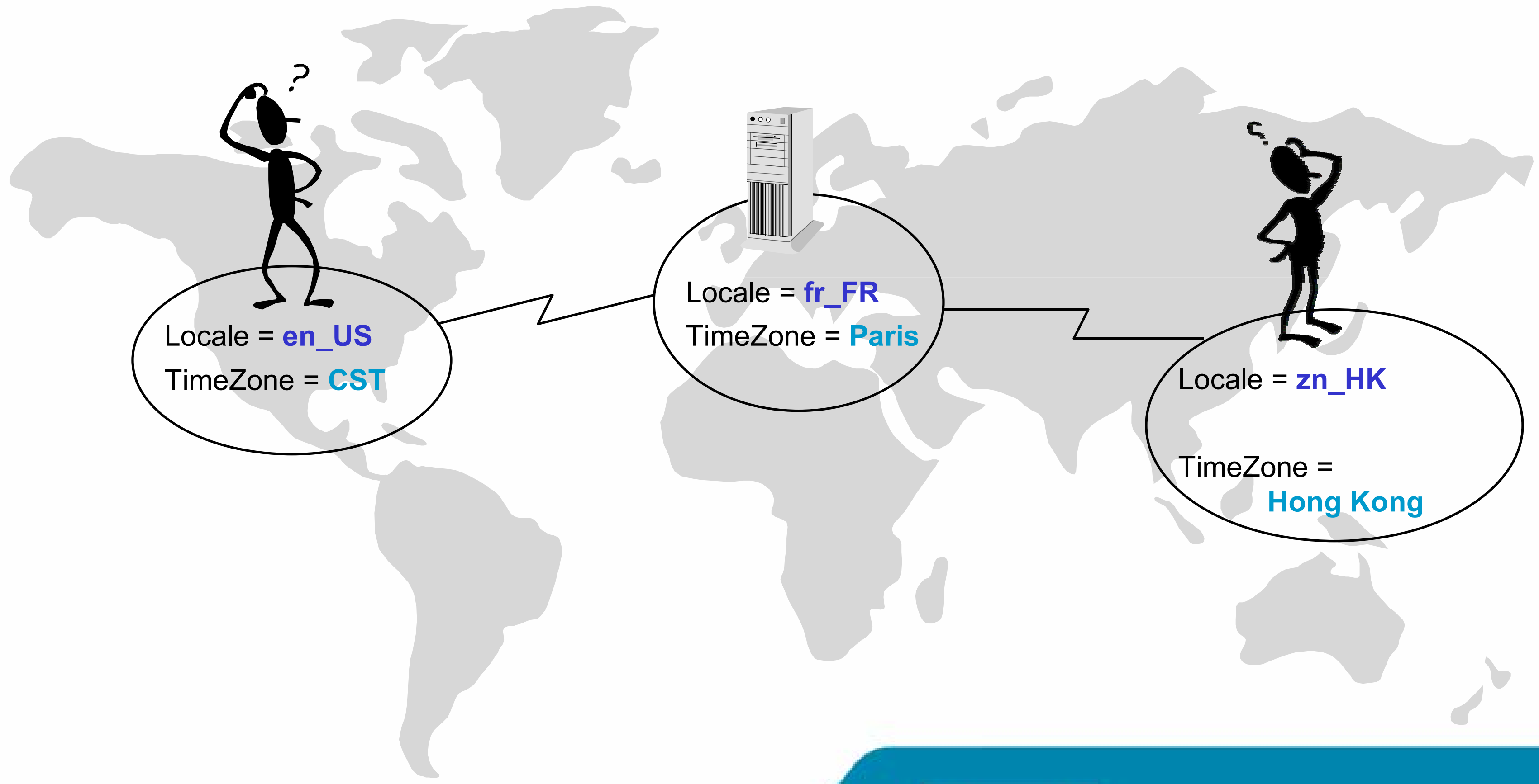
Tivoli

18th International Unicode Conference

Hong Kong, April 2001

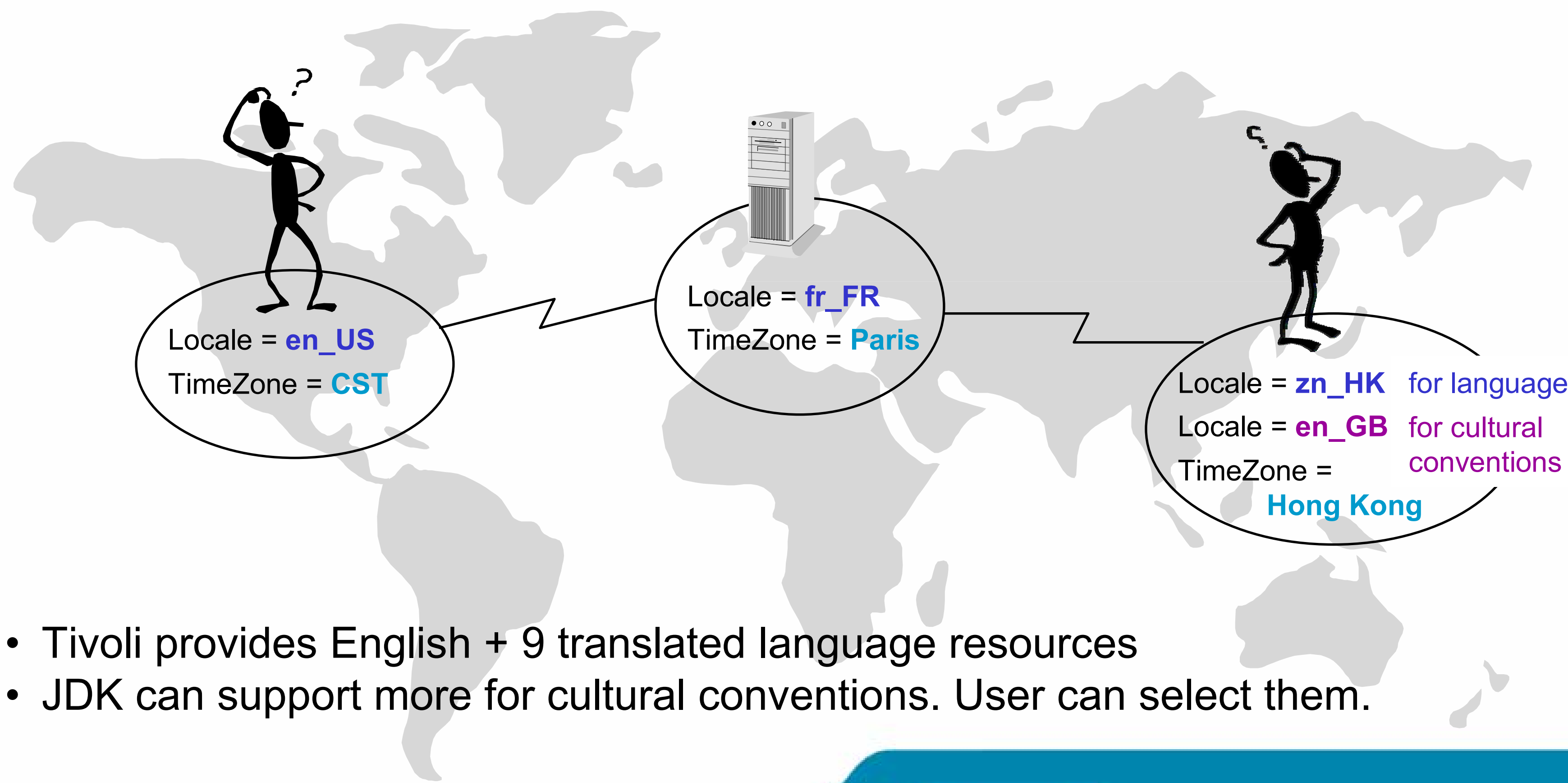
Req.1: Global multi-user support

- Support multiple users simultaneously across different locales and time zones



Req.2: Multiple locales for a single user

- Allow user to select different locales for resource lookup and cultural conventions



- Tivoli provides English + 9 translated language resources
- JDK can support more for cultural conventions. User can select them.

Req.2: Multiple locales for a single user

- Locale constructor

```
Locale(String language, String country)

Locale loc = new Locale(“fr”, “FR”);     // French (France)

Locale loc = new Locale(“el”, “GR”);     // Greek (Greece)

Locale loc = new Locale(“fr”, “GR”);     // French (Greece) ?
```

Tivoli

Req.3: End-user Customization

- Allow user to override a locale's default attributes
  - e.g. date format styles and symbols
- The preferred non-default style should be persisted and to be used for future session, by all framework applications

Req.4: Advanced features

- More features than the JDK
  - Multi-cultural calendar (Gregorian, Buddhist, Japanese, Hebrew, Islamic, Chinese)
  - Some other requirements are being discussed for future release. For example, transliteration, Java input method editor, etc.

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
  - JDK's Internationalization API
  - Example
- Approach
- Conclusion
- Q&A

Difficulties with JDK

JDK’s Internationalization API

- Rich features, highly customizable
- Locale-sensitive classes in
  - java.util package
  - java.text package
  - etc.
- One default locale can be set

Example: set default locale

```
// set default locale
Locale.setDefault(aLocale);
```

Example 1: format a date

```
// everything is based on the default locale

DateFormat df = DateFormat.getDateTimeInstance();
String dateDisplay = sdf.format(aDate);
```

Example 1: format a date

```
// use cultLocale

DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT,
                                               cultLocale);
String dateDisplay = sdf.format(aDate);
```

Example 1: format a date

```
// use cultLocale, dateTimePattern

DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT,
                                               cultLocale);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
String dateDisplay = sdf.format(aDate);
```

Example 1: format a date

```
// use cultLocale, dateTimePattern, ampmStr



DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT,
                                               cultLocale);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
DateFormatSymbols dfs = sdf.getDateFormatSymbols();
dfs.setAmPmStrings(ampmStr);
sdf.setDateFormatSymbols(dfs);
String dateDisplay = sdf.format(aDate);
```

Example 1: format a date

```
// use cultLocale, dateTimePattern, ampmStr, timeZoneID

TimeZone tz = TimeZone.getTimeZone(timeZoneID);
Calendar cal = Calendar.getInstance(tz, cultLocale);
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT,
                                               cultLocale);
df.setCalendar(cal);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
DateFormatSymbols dfs = sdf.getDateFormatSymbols();
dfs.setAmPmStrings(ampmStr);
sdf.setDateFormatSymbols(dfs);
String dateDisplay = sdf.format(aDate);
```

Tivoli

Example 2: format a message

```
// use langLocale, cultLocale, dateTimePattern, ampmstr, timeZoneID
TimeZone tz = TimeZone.getTimeZone(timeZoneID);
Calendar cal = Calendar.getInstance(tz, cultLocale);
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT, cultLocale);
df.setCalendar(cal);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
DateFormatSymbols dfs = sdf.getDateFormatSymbols();
dfs.setAmPmStrings(ampmStr);
sdf.setDateFormatSymbols(dfs);
ResourceBundle rb = ResourceBundle.getBundle("MessageResources", langLocale);
String msgPattern = rb.getString(MessageResources.MSG0001);
MessageFormat mf = new MessageFormat(msgPattern);
mf.setLocale(cultLocale);
mf.applyPattern(msgPattern);
mf.setFormat(0, sdf);
String msg = mf.format(new Object[] {aDate});
```

Tivoli

How can we . . .

- fulfill requirements for internationalization and
- keep application’s code simple as much as possible and
- keep consistent result across various applications on the framework?

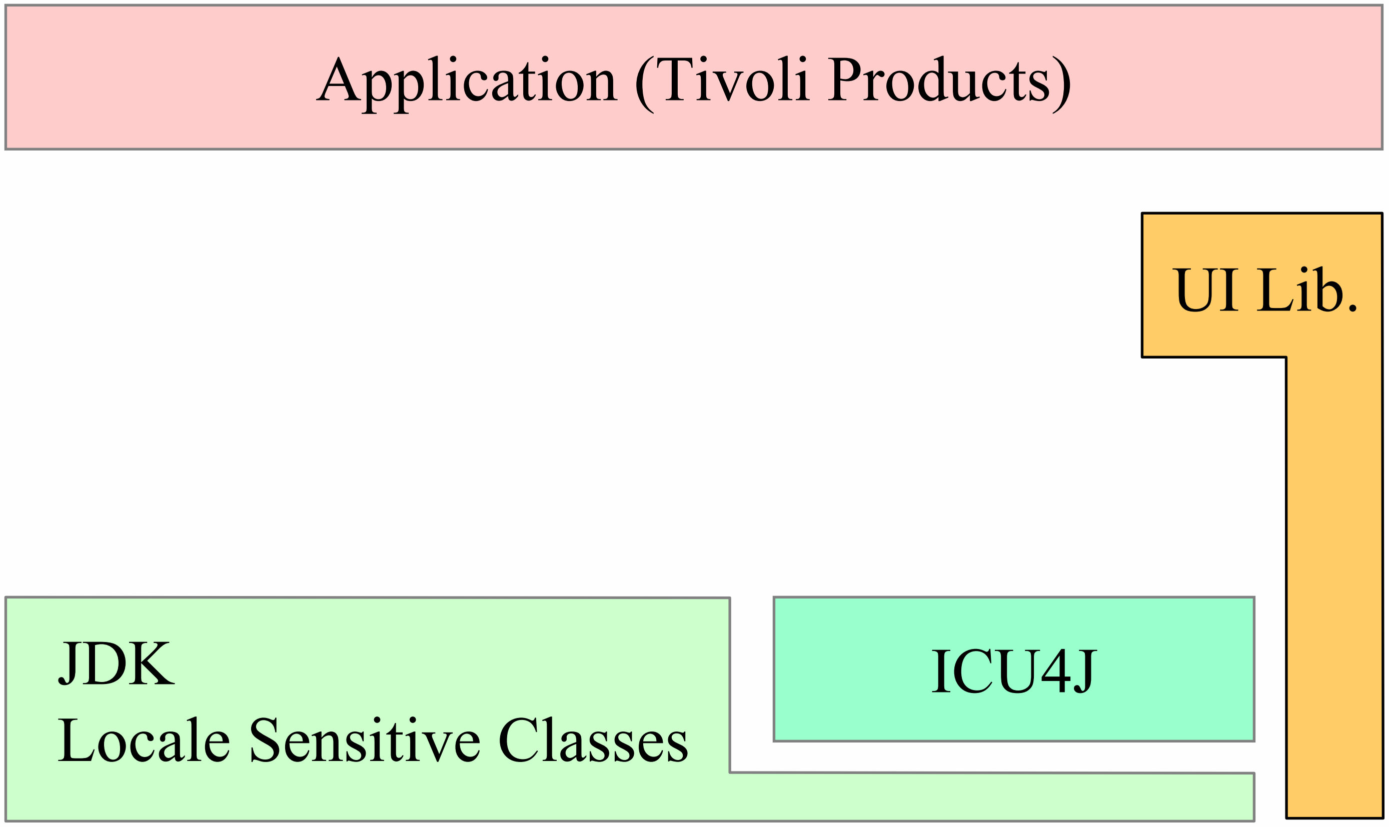
Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
  - High-level Framework Service
- Conclusion
- Q&A

Approach

Tivoli

Hierarchy of Internationalization APIs



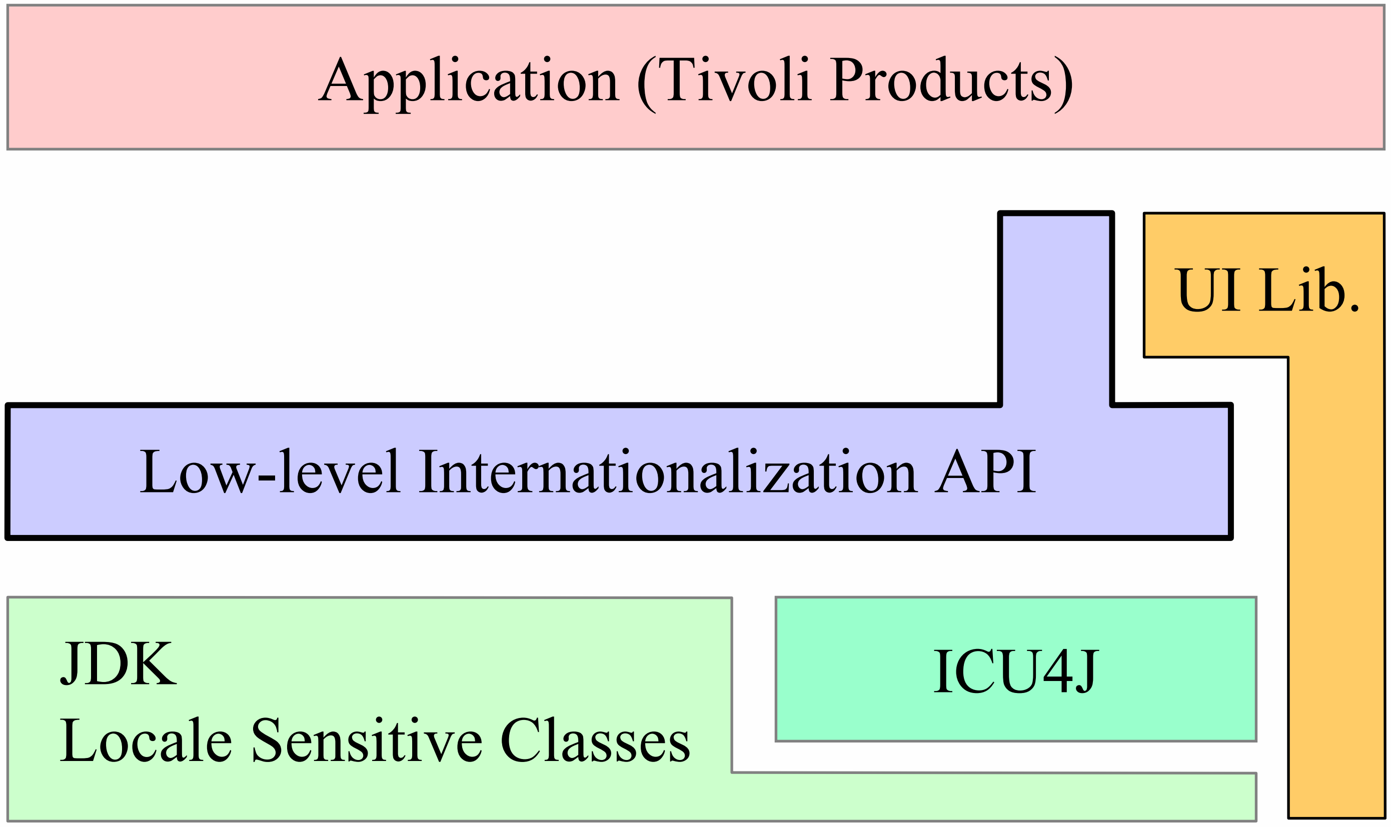
ICU4J: International Components for Unicode for Java

Use of ICU4J

ICU4J is . . .

- International Components for Unicode for Java
- IBM’s open source project
- Java classes for internationalization in Unicode
  - International Calendars
  - Unicode Normalization
  - Number Format Enhancements
  - Enhanced word-break detection
  - Unicode Text Searching
  - Unicode Text Compression
  - etc...
- http://oss.software.ibm.com/developerworks/opensource/icu4j/

Hierarchy of Internationalization APIs


Application (Tivoli Products)
UI Lib.
Low-level Internationalization API
JDK
Locale Sensitive Classes
ICU4J

ICU4J: International Components for Unicode for Java

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
    - International Preference Object ⬅
    - JDK/ICU4J Wrapper
    - Other classes
    - Rewrite Example
  - High-level Framework Service
- Conclusion
- Q&A

Tivoli

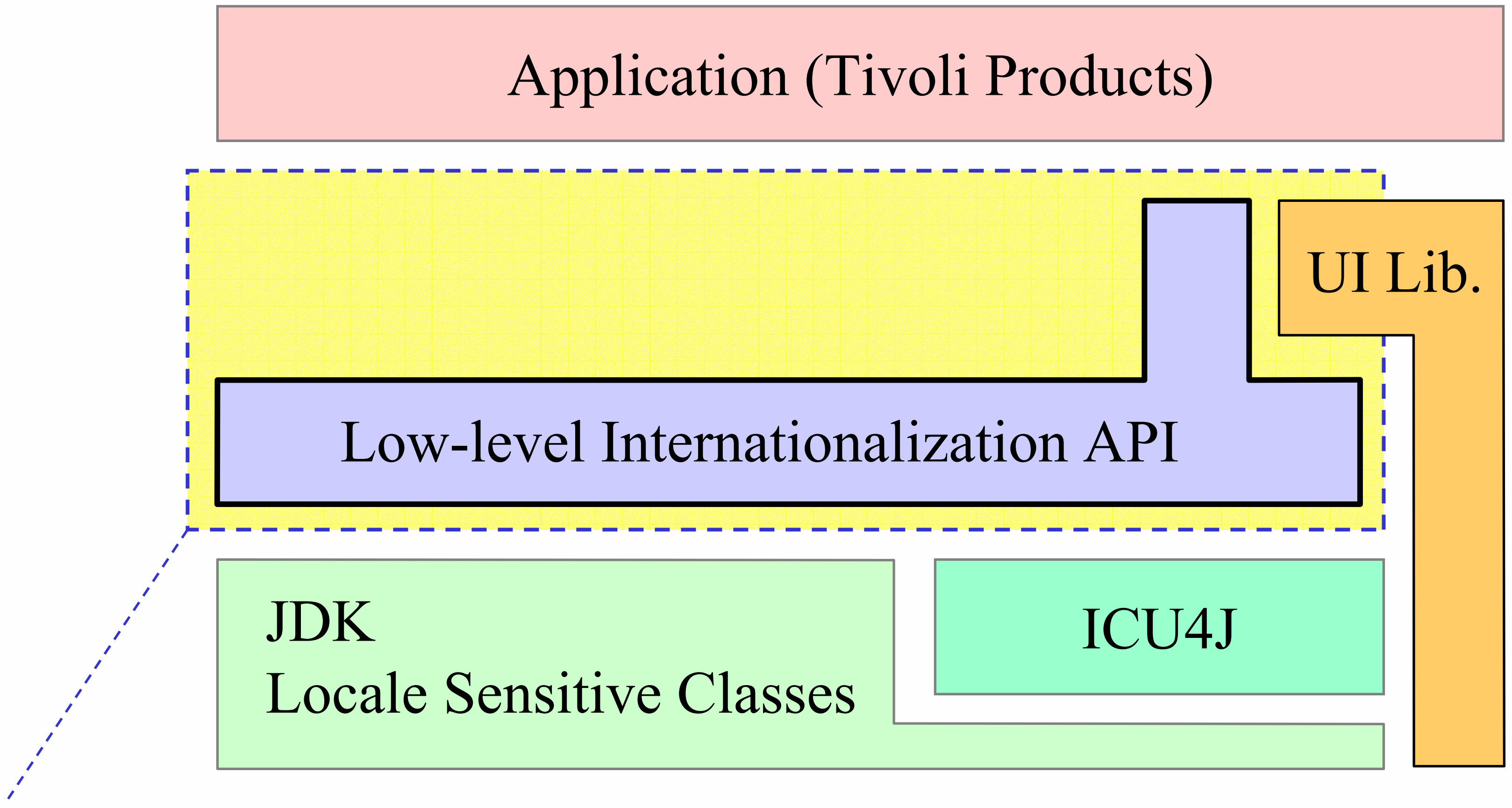
International Preference Object

- Encapsulates user’s internationalization preferences
  - Language (Locale for resource lookup, HelpSet access, etc.)
  - Region (Locale for data formatting, etc.)
  - Calendar Type (Gregorian, Buddhist, Japanese, Islamic, Hebrew, Chinese)
  - Time Zone
  - Default override for Number format patterns and symbols
  - Default override for Date/Time format patterns and symbols
- All low-level API work with this object

Setters on IntlPreferences Class

- setLanguageLocale
- setCulturalConventionLocale
- setTimeZone
- setCalendarType
- setDefaultDateStyle
- setDatePattern
- setDefaultTimeStyle
- setTimePattern
- setDateTimeOrder
- setDateTimeSeparator
- setAMString
- setPMString
- setNumberPattern
- setCurrencyPattern
- setPercentPattern
- setGroupingSeparator
- setDecimalSeparator
- setPercent
- setZeroDigit
- setMinusSign
- setCurrencySymbol
- setInternationalCurrencySymbol
- setMonetaryDecimalSeparator

Hierarchy of Internationalization APIs


Application (Tivoli Products)
UI Lib.
Low-level Internationalization API
JDK
Locale Sensitive Classes
ICU4J

International Preference-based API

ICU4J: International Components for Unicode for Java

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - – Use of ICU4J
  - – Low-level Internationalization API
    - International Preference Object
    - JDK/ICU4J Wrapper ⬅
    - Other classes
    - Rewrite Example
  - – High-level Framework Service
- Conclusion
- Q&A

JDK/ICU4J Wrapper

Purpose:

- Extend locale-based API to the international preferences object-based API
- Add one layer on ICU4J to avoid side effects of possible API changes

JDK/ICU4J Wrapper Classes

- com.tivoli.intl

  IntlTimeZone
  IntlSimpleTimeZone
  IntlCalendar
  IntlGregorianCalendar
  IntlBuddhistCalendar
  IntlJapaneseCalendar
  IntlHebrewCalendar
  IntlIslamicCalendar
  IntlChineseCalendar

- com.tivoli.intl

  IntlDateFormat
  IntlSimpleDateFormat
  IntlDateFormatSymbols
  IntlNumberFormat
  IntlDecimalFormat
  IntlDecimalFormatSymbols
  IntlMessageFormat

Tivoli

JDK/ICU4J Wrapper Class Example

java.text.DateFormat class (JDK)

com.ibm.text.DateFormat class (ICU4J)

```
public static DateFormat getDateInstance()
public static DateFormat getDateInstance(int style)
public static DateFormat getDateInstance(int style, Locale locale)
```

com.tivoli.intl.IntlDateFormat class (Tivoli Wrapper)

```
public static IntlDateFormat getDateInstance()
public static IntlDateFormat getDateInstance(int style)
public static IntlDateFormat getDateInstance(int style, Locale locale)
public static IntlDateFormat getDateInstance(IntlPreferences ip)
```

Other Classes

- com.tivoli.intl

IntlPreferences
IntlUtilities
IntlDisplayableText
IntlBundleLoader

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
    - International Preference Object
    - JDK/ICU4J Wrapper
    - Other classes
    - Rewrite Example ⬅
  - High-level Framework Service
- Conclusion
- Q&A

Example 1: format a date

```
// use cultLocale, dateTimePattern, ampmStr, timeZoneID

TimeZone tz = TimeZone.getTimeZone(timeZoneID);
Calendar cal = Calendar.getInstance(tz, cultLocale);
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                               DateFormat.DEFAULT,
                                               cultLocale);
df.setCalendar(cal);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
DateFormatSymbols dfs = sdf.getDateFormatSymbols();
dfs.setAmPmStrings(ampmStr);
sdf.setDateFormatSymbols(dfs);
String dateDisplay = sdf.format(aDate);
```

Example 2: format a message

```
// use langLocale, cultLocale, dateTimePattern, ampmstr, timeZoneID
TimeZone tz = TimeZone.getTimeZone(timeZoneID);
Calendar cal = Calendar.getInstance(tz, cultLocale);
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.DEFAULT,
                                              DateFormat.DEFAULT, cultLocale);
df.setCalendar(cal);
SimpleDateFormat sdf = (SimpleDateFormat)df;
sdf.applyPattern(dateTimePattern);
DateFormatSymbols dfs = sdf.getDateFormatSymbols();
dfs.setAmPmStrings(ampmStr);
sdf.setDateFormatSymbols(dfs);
ResourceBundle rb = ResourceBundle.getBundle("MessageResources", langLocale);
String msgPattern = rb.getString(MessageResources.MSG0001);
MessageFormat mf = new MessageFormat(msgPattern);
mf.setLocale(cultLocale);
mf.applyPattern(msgPattern);
mf.setFormat(0, sdf);
String msg = mf.format(new Object[] {aDate});
```

Rewrite examples: Customization

```
// Customizations are centralized to the International preferences

ip.setLanguageLocale(langLocale);
ip.setCulturalConventionLocale(cultLocale);
ip.setTimeZone(timeZoneID);
ip.setAMString(ampmStr[0]);
ip.setPMString(ampmStr[1]);
ip.setDatePattern(datePattern, IntlPreferences.MEDIUM);
ip.setTimePattern(timePattern, IntlPreferences.MEDIUM);
ip.setDateTimeOrder(IntlPreferences.DATE_FIRST);
                                . . .
```

(. . . 23 setters can be used)

Rewrite example 1: format a date

```
// everything is based on the default locale

DateFormat df = DateFormat.getDateTimeInstance();
String dateDisplay = df.format(aDate);
```

```
// use international preferences

IntlDateFormat df = IntlDateFormat.getDateTimeInstance(ip);
String dateDisplay = df.format(aDate);
```

Rewrite example 2: format a message

```
// everything is based on the default locale

ResourceBundle rb = ResourceBundle.getBundle("MessageResources");
String msgPattern = rb.getString(MessageResources.MSG0001);
MessageFormat mf = new MessageFormat(msgPattern);
String msg = mf.format(new Object[] {aDate});
```

↓

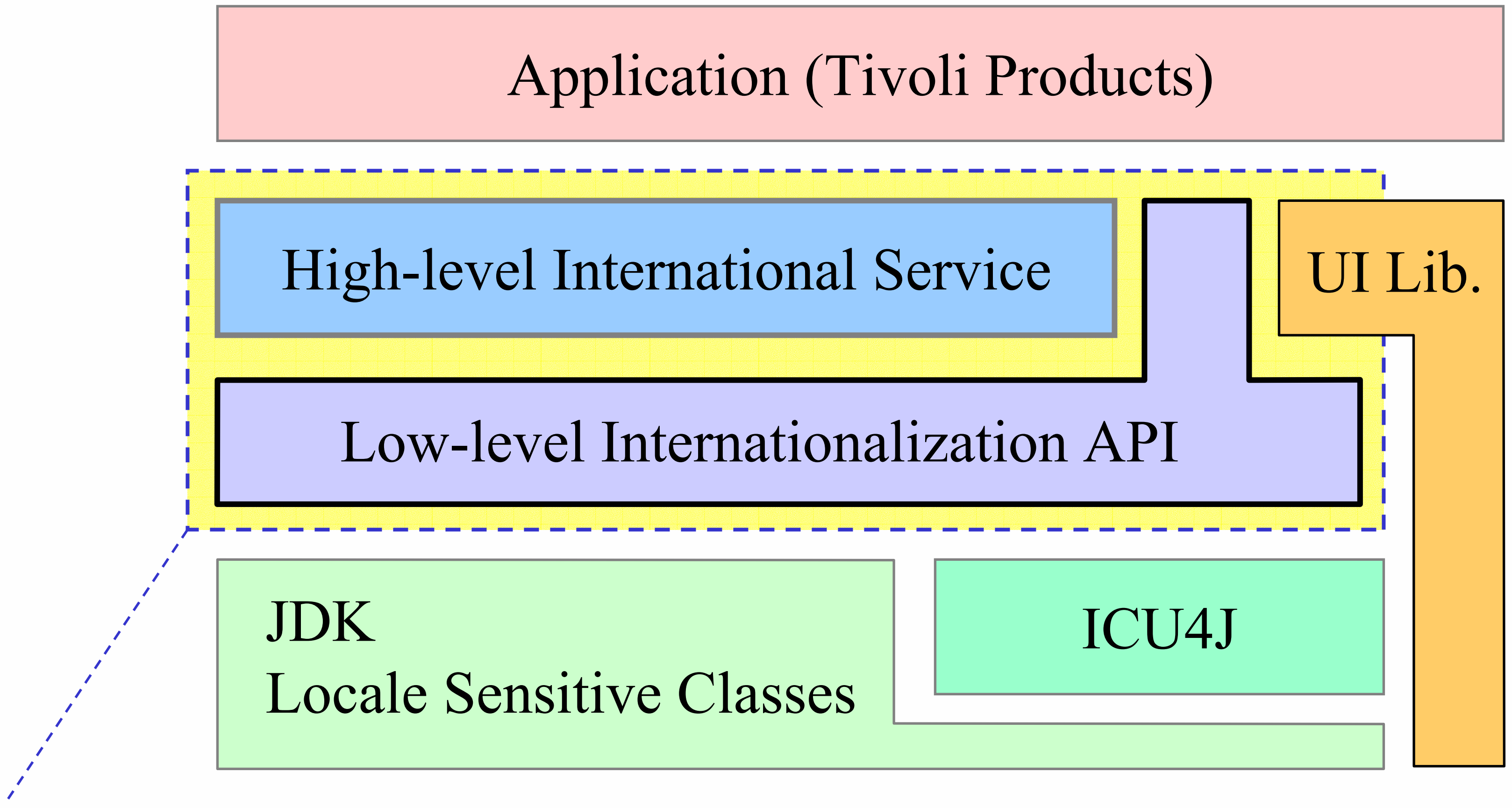
```
// use international preferences

ResourceBundle rb = IntlUtilities.getBundle(ip, "MessageResources");
String msgPattern = rb.getString(MessageResources.MSG0001);
IntlMessageFormat mf = new IntlMessageFormat(ip, msgPattern);
String msg = mf.format(new Object[] {aDate});
```

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
  - High-level Framework Service
    - Layer Separation ⬅
    - High-level Framework Service
    - Rewrite Example
    - International Preferences Notebook
    - Example
- Conclusion
- Q&A

Tivoli

Hierarchy of Internationalization APIs


Application (Tivoli Products)
High-level International Service
UI Lib.
Low-level Internationalization API
JDK
Locale Sensitive Classes
ICU4J

International Preference-based API

ICU4J: International Components for Unicode for Java

Layer Separation

High-level International Service

Include the framework specific implementations

Correlate application's context to the user's international preference

Be generic for reusability

Take international preference object from method argument

Low-level Internationalization API

Agenda

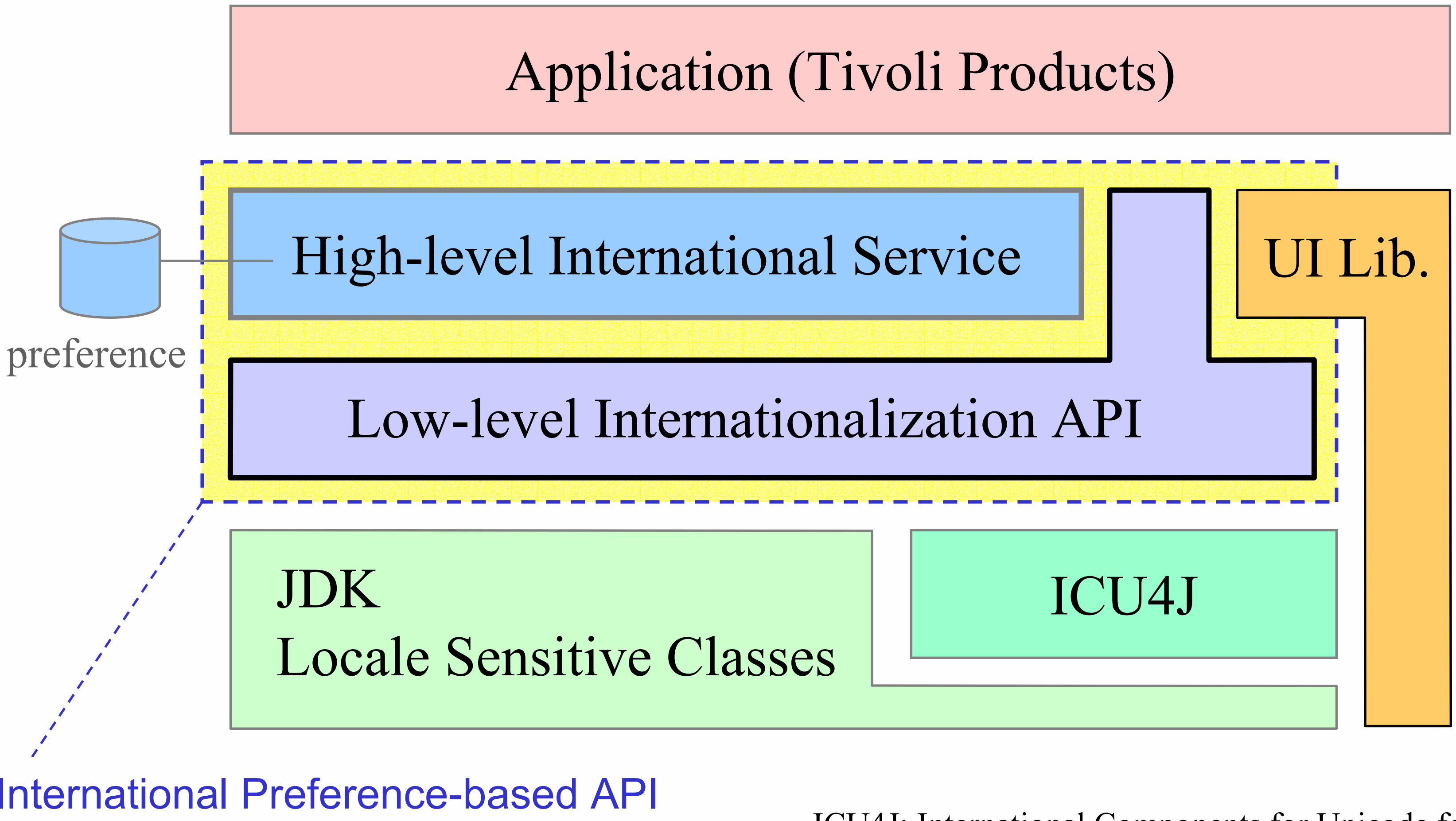
- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
  - High-level Framework Service
    - Layer Separation
    - High-level Framework Service ⬅
    - Rewrite Example
    - International Preferences Notebook
    - Example
- Conclusion
- Q&A

Tivoli

High-level International Service

- One of the services for the framework applications
- Provides a set of convenience methods on top of low-level internationalization API
- Manages the international preferences object according to the application context
- The preferences object is persisted using a framework service.

Hierarchy of Internationalization APIs


Application (Tivoli Products)
High-level International Service
preference
UI Lib.
Low-level Internationalization API
JDK
Locale Sensitive Classes
ICU4J
International Preference-based API

ICU4J: International Components for Unicode for Java

Methods on International Service

- **com.tivoli.pf.fmk.external.IFmkIntlService**
  - Resource Loading
    - **getBundle, getString, getObject**
  - Date Formatting
    - **formatDate, formatTime, formatDateTime, parseDate, parseTime, parseDateTime**
  - Number Formatting
    - **formatNumber, formatCurrency, formatPercent, parseNumber, parseCurrency, parsePercent**
  - Message Formatting
    - **formatMessage, parseMessage**
  - Collator Factory Method
    - **getCollator**
  - BreakIterator Factory Method
    - **getCharacterBreakIterator, getWordBreakIterator, getLineBreakIterator, getSentenceBreakIterator**
  - Case Conversion / Comparison
    - **toLowerCase, toUpperCase, equalsIgnoreCase**
  - Late-binding Text Resolution
    - **getDisplayText**
  - IntlPreferences Access
    - **getIntlPreferences**

Tivoli

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
  - High-level Framework Service
    - Layer Separation
    - High-level Framework Service
    - Rewrite Example
    - International Preferences Notebook
    - Example
- Conclusion
- Q&A

Rewrite example 1: format a date

```
// everything is based on the default locale

DateFormat df = DateFormat.getDateTimeInstance();
String dateDisplay = df.format(aDate);
```

```
// works according to the user's international preferences

String dateDisplay = intlService.formatDate(aContext, aDate);
```

Rewrite example 2: format a message

```
// everything is based on the default locale

ResourceBundle rb = ResourceBundle.getBundle("MessageResources");
String msgPattern = rb.getString(MessageResources.MSG0001);
MessageFormat mf = new MessageFormat(msgPattern);
String msg = mf.format(new Object[] {aDate});
```

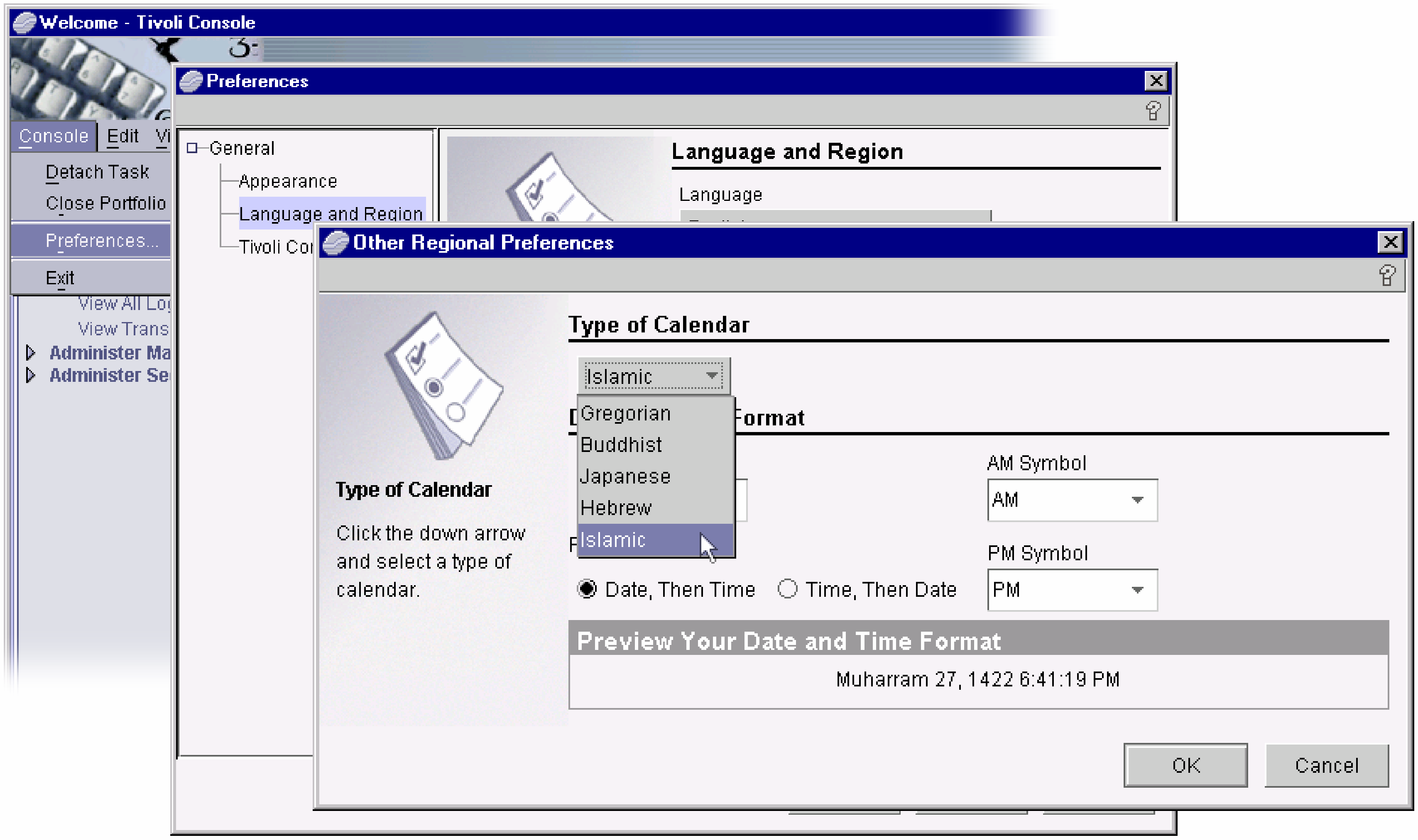
$\Downarrow$

```
// works according to the user's international preferences

String msg = intlService.getString(aContext,
                                   "MessageResources",
                                    MessageResources.MSG0001,
                                    new Object[] {aDate});
```

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - Use of ICU4J
  - Low-level Internationalization API
  - High-level Framework Service
    - Layer Separation
    - High-level Framework Service
    - Rewrite Example
    - International Preferences Notebook ⬅
    - Example
- Conclusion
- Q&A

International Preference Notebook



Type of Calendar

Click the down arrow and select a type of calendar.

Type of Calendar

Islamic

AM Symbol: AM

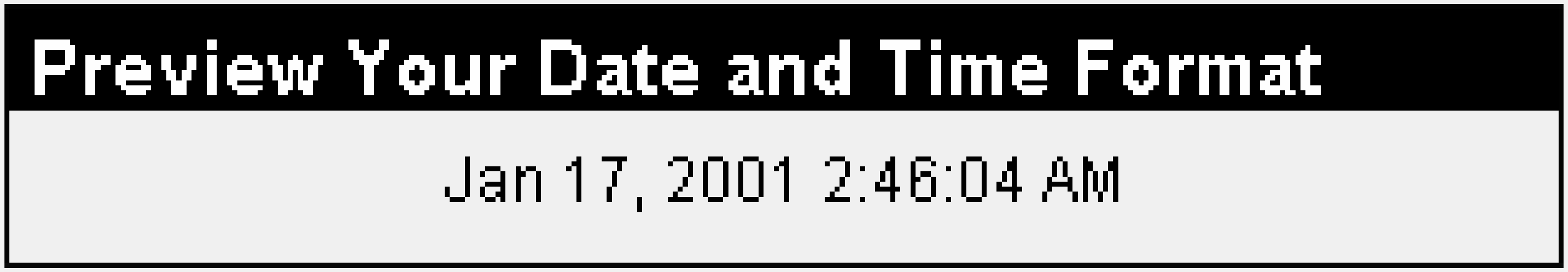
PM Symbol: PM

[✓ Date, Then Time] Time, Then Date

Preview Your Date and Time Format

Muharram 27, 1422 6:41:19 PM

Preference Notebook Preview Example

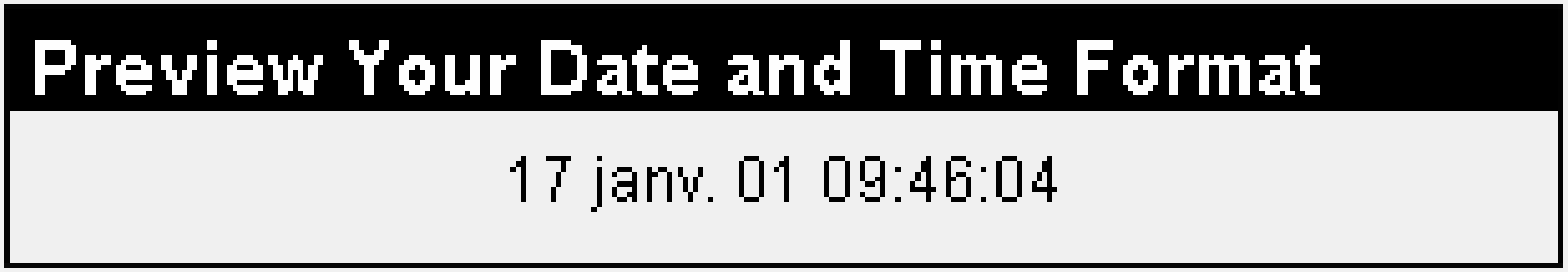


Preview Your Date and Time Format

Jan 17, 2001 2:46:04 AM

Region:	US
Style:	Medium
Calendar:	Gregorian
Time Zone:	CST

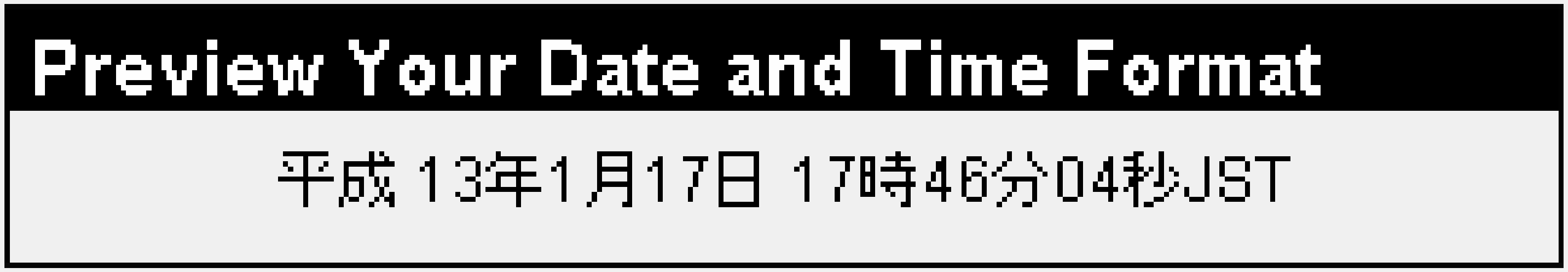
Preference Notebook Preview Example


Preview Your Date and Time Format

17 janv. 01 09:46:04

Region:	France
Style:	Medium
Calendar:	Gregorian
Time Zone:	Paris

Preference Notebook Preview Example



Preview Your Date and Time Format

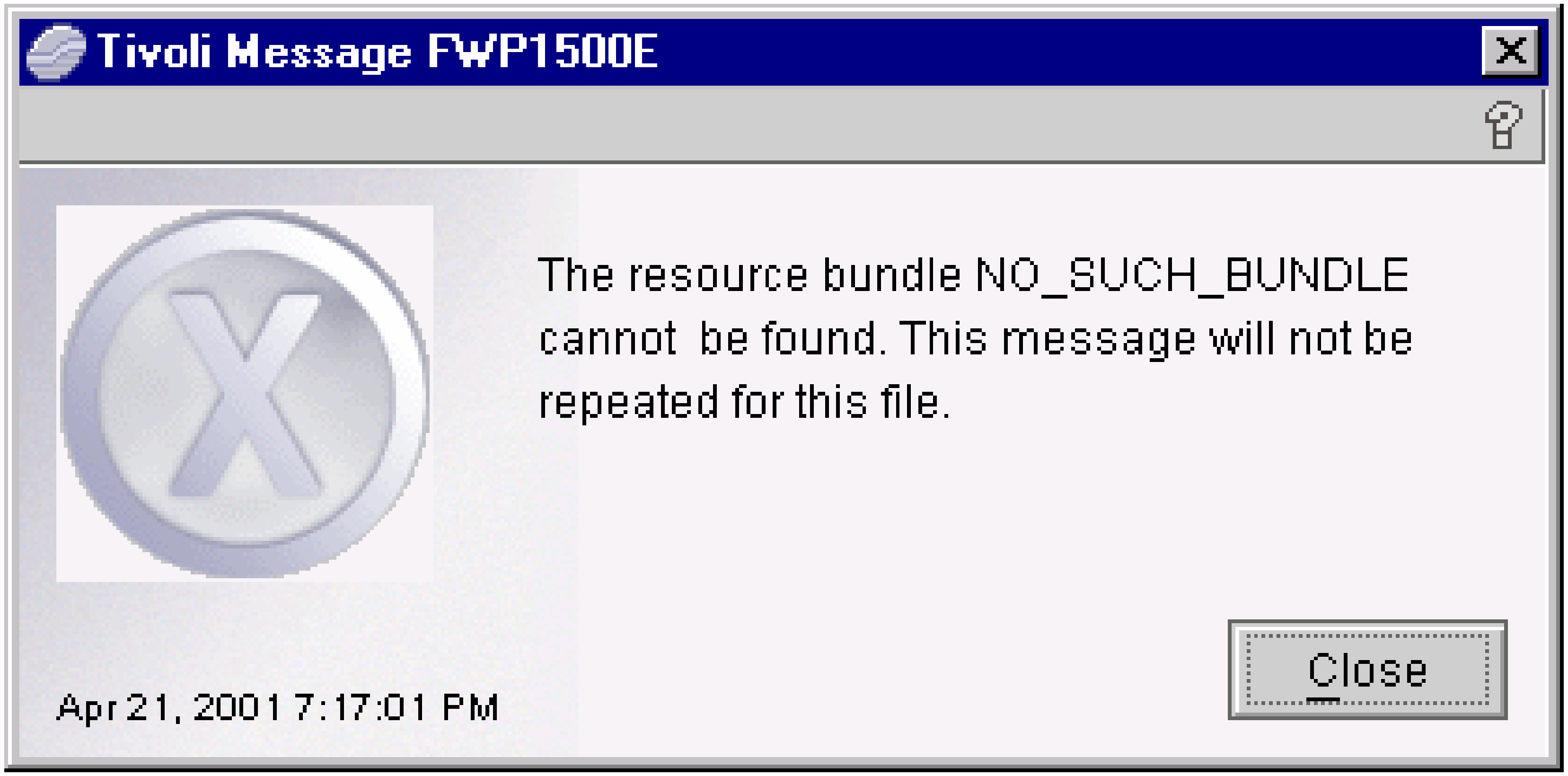
平成 13年1月17日 17時46分04秒JST

Region:	Japan
Style:	Full
Calendar:	Japanese-era
Time Zone:	Tokyo

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
  - – Use of ICU4J
  - – Low-level Internationalization API
  - – High-level Framework Service
    - Layer Separation
    - High-level Framework Service
    - Rewrite Example
    - International Preferences Notebook
    - Example ⇐
- Conclusion
- Q&A

Example – Message Dialog

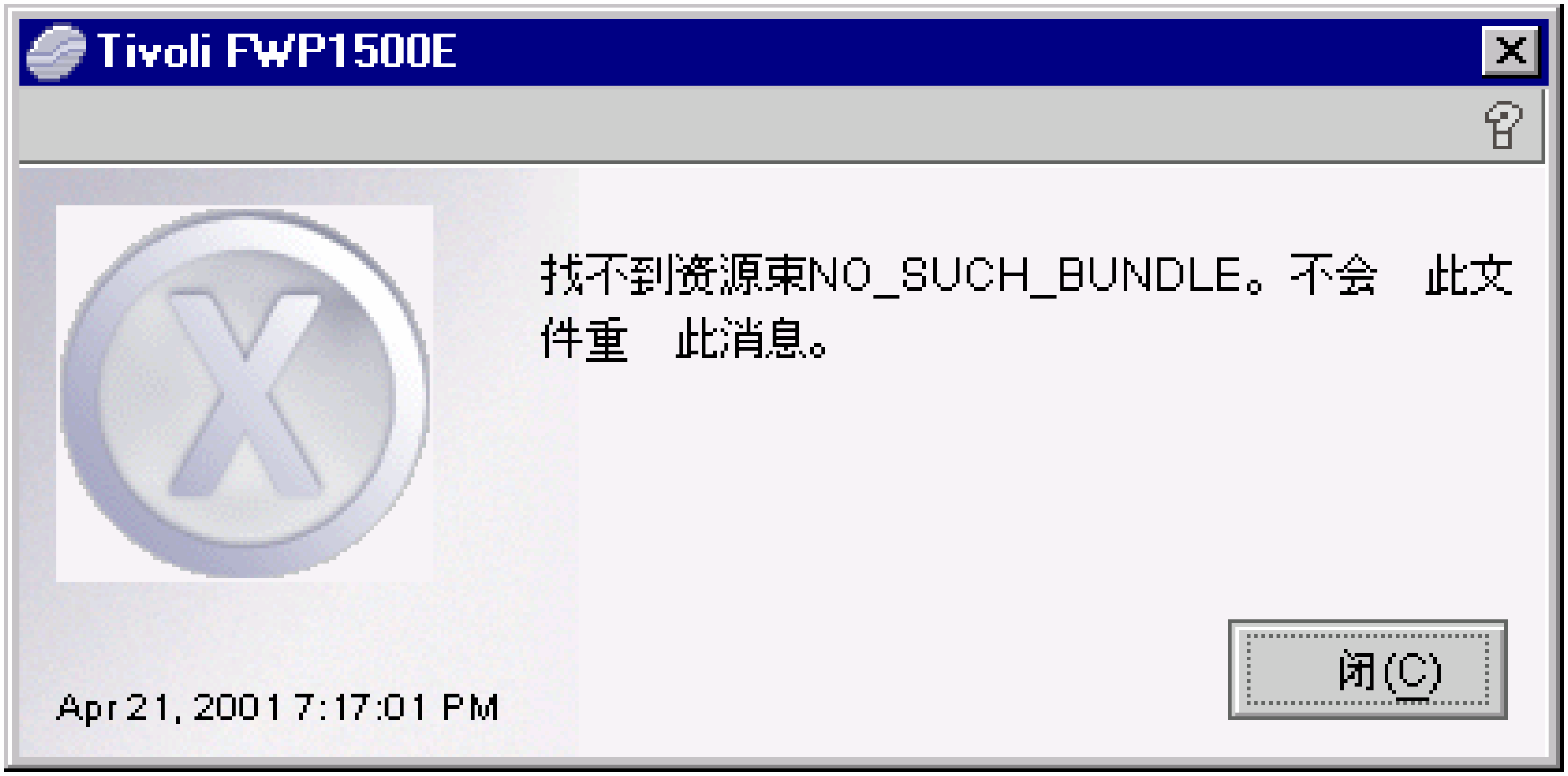

Tivoli Message FWP1500E

The resource bundle NO_SUCH_BUNDLE cannot be found. This message will not be repeated for this file.

Apr 21, 2001 7:17:01 PM

Language: English
Region: US

Example – Message Dialog


Tivoli FWP1500E

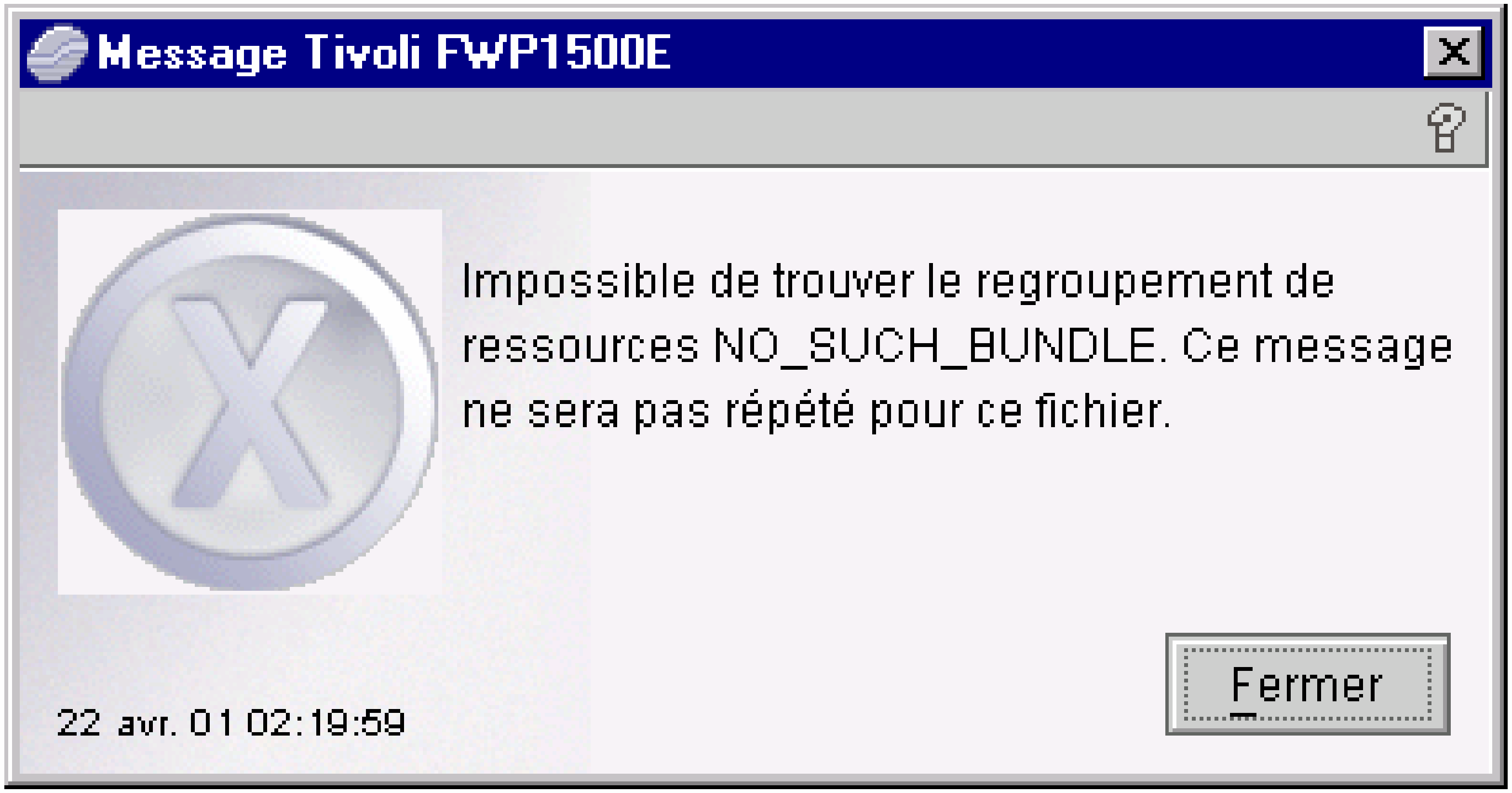
找不到资源束NO_SUCH_BUNDLE。不会　此文件重　此消息。

Apr 21, 2001 7:17:01 PM

Language: Simplified Chinese

Region: US

Example – Message Dialog



Message Tivoli FWP1500E

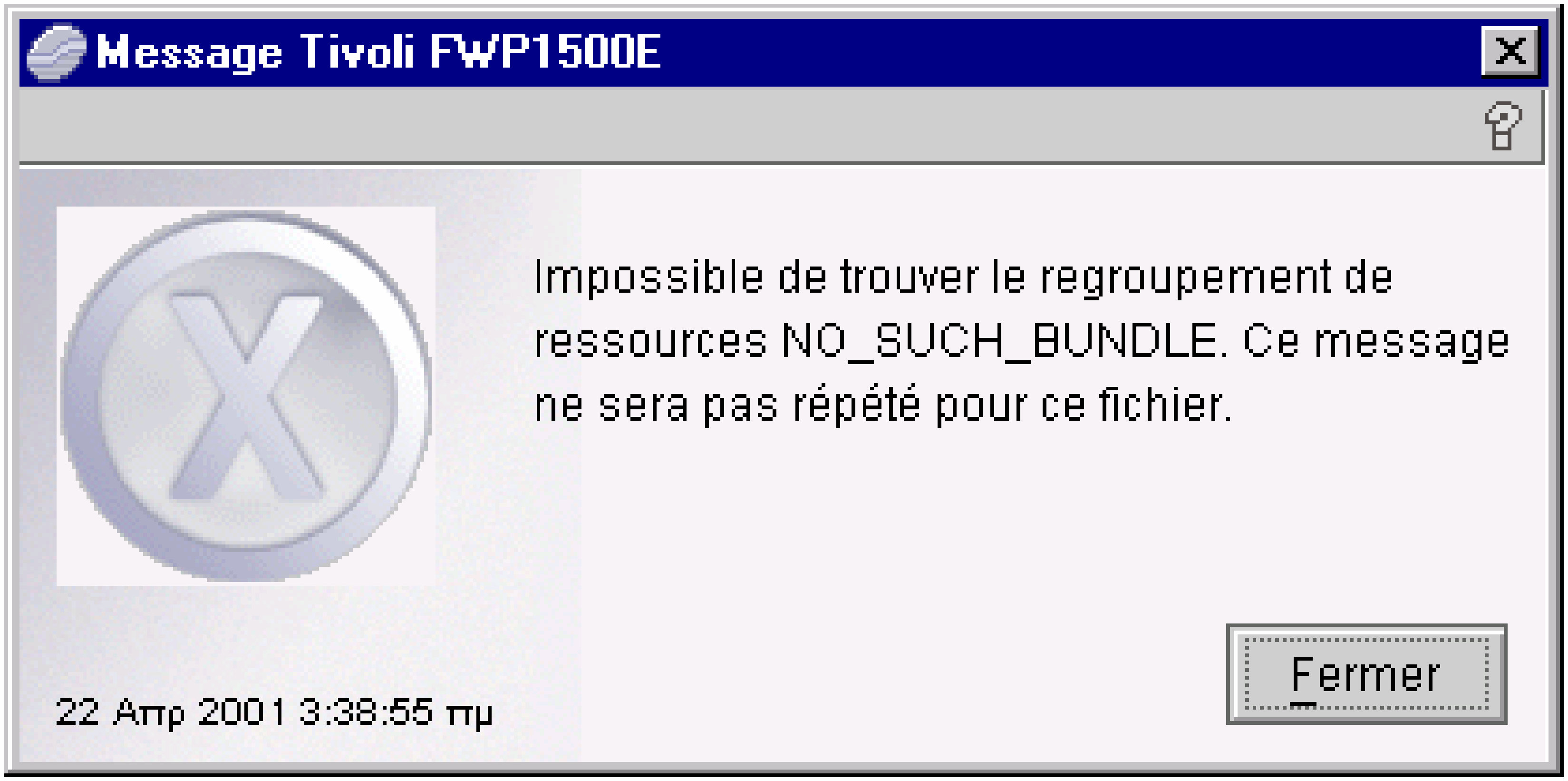
Impossible de trouver le regroupement de ressources NO_SUCH_BUNDLE. Ce message ne sera pas répété pour ce fichier.

22 avr. 01 02:19:59

Language: French

Region: France

Example – Message Dialog



Message Tivoli FWP1500E

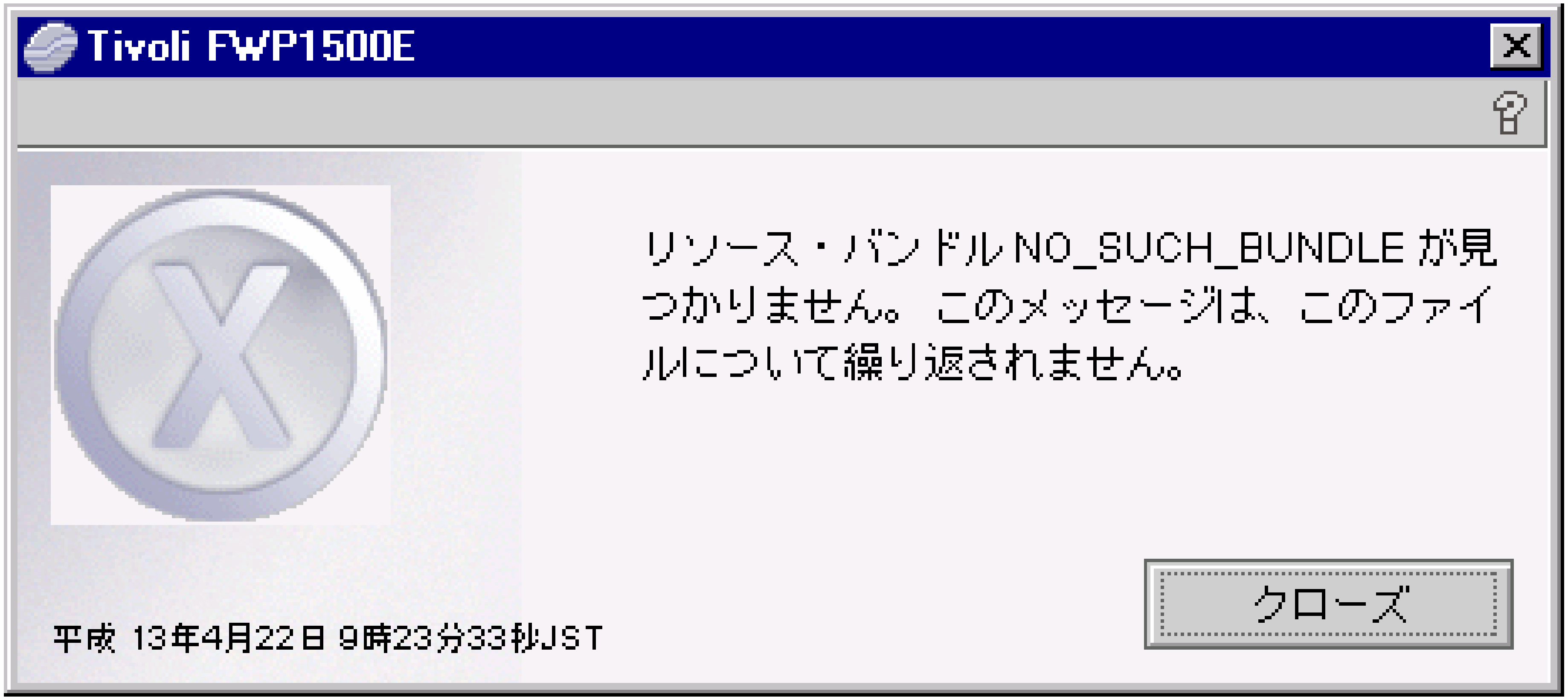
Impossible de trouver le regroupement de ressources NO_SUCH_BUNDLE. Ce message ne sera pas répété pour ce fichier.

22 Απρ 2001 3:38:55 πμ

Language: French

Region: Greece

Example – Message Dialog



Tivoli FWP1500E

リソース・バンドル NO_SUCH_BUNDLE が見つかりません。 このメッセージは、このファイルについて繰り返されません。

平成 13年4月22日 9時23分33秒JST

Language:	Japanese
Region:	Japan
Calendar:	Japanese

Agenda

- Introduction
- Requirements for Internationalization
- Difficulties with JDK
- Approach
- Conclusion ⬅
  - For Requirements…
    1. Global multi-user support
    2. Multiple locales for a single user
    3. End-user Customization
    4. Advanced Features
  - Benefits
- Q&A

Tivoli

Conclusion

For req.1: Global multi-user support

- In high-level international service layer, the user’s locale and time zone is managed according to the application context.
- Each user’s preference settings are persistent across the network.

Tivoli

For req.2: Multiple locales for a single user

- Multiple locales are encapsulated in the international preferences object.
- Low-level API uses appropriate locale from the preferences object.

For req.3: End-user Customization

- Various attributes for customization encapsulated in an object. It simplifies API usage.
- User can alter the object through preference notebook GUI.

For req.4: Advanced features

- Multi-cultural calendar systems are supported
- ICU4J is used as core internationalization API. More feature will be used in the future releases.

Benefits for end-users

- Flexible customization
- Functionality for multiple-locale
- Consistent presentation

Benefits for application developer

- High-level International Service can be used as a single point to provide internationalization API
- Various internationalization attributes are encapsulated into an object to make API usage simple
- International preference-based Swing widgets

Questions & Answers

Tivoli