19

Localization

ORACLE

Objectives

After completing this lesson, you should be able to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle

Why Localize?

The decision to create a version of an application for international use often happens at the start of a development project.

- Region- and language-aware software
- Dates, numbers, and currencies formatted for specific countries
- Ability to plug in country-specific data without changing code

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

In addition to language changes, culturally dependent elements, such as dates, numbers, currencies, and so on must be translated.

The goal is to design for localization so that no coding changes are required.

A Sample Application

Localize a sample application:

- Text-based user interface
- Localize menus
- Display currency and date localizations

```
=== Localization App ===
1. Set to English
2. Set to French
3. Set to Chinese
4. Set to Russian
5. Show me the date
6. Show me the money!
q. Enter q to quit
Enter a command:
```

In the remainder of this lesson, this simple text-based user interface will be localized for French, Simplified Chinese, and Russian. Enter the number indicated by the menu and that menu option will be applied to the application. Enter `q` to exit the application.

Locale

A `Locale` specifies a particular language and country:

- Language
  - An alpha-2 or alpha-3 ISO 639 code
  - “en” for English, “es” for Spanish
  - Always uses lowercase
- Country
  - Uses the ISO 3166 alpha-2 country code or UN M.49 numeric area code
  - "US" for United States, "ES" for Spain
  - Always uses uppercase
- See the Java Tutorials for details of all standards used.

In Java, a locale is specified by using two values: language and country. See the Java Tutorial for standards used:
`http://download.oracle.com/javase/tutorial/i18n/locale/create.html`

Language Samples

- de: German
- en: English
- fr: French
- zh: Chinese

Country Samples

- DE: Germany
- US: United States
- FR: France
- CN: China

Properties

- The `java.util.Properties` class is used to load and save key-value pairs in Java.
- Can be stored in a simple text file:

```
hostName = www.example.com
userName = user
password = pass
```

- File name ends in `.properties`.
- File can be anywhere that compiler can find it.

The benefit of a properties file is the ability to set values for your application externally. The properties file is typically read at the start of the application and is used for default values. But the properties file can also be an integral part of a localization scheme, where you store the values of menu labels and text for various languages that your application may support.

The convention for a properties file is `<filename>.properties`, but the file can have any extension you want. The file can be located anywhere that the application can find it.

Loading and Using a Properties File

```
1   public static void main(String[] args) {
2     Properties myProps = new Properties();
3     try {
4       FileInputStream fis = new FileInputStream("ServerInfo.properties");
5       myProps.load(fis);
6     } catch (IOException e) {
7       System.out.println("Error: " + e.getMessage());
8     }
9
10     // Print Values
11     System.out.println("Server: " + myProps.getProperty("hostName"));
12     System.out.println("User: " + myProps.getProperty("userName"));
13     System.out.println("Password: " + myProps.getProperty("password"));
14  }
```


In the code fragment, you create a `Properties` object. Then, using a `try` statement, you open a file relative to the source files in your NetBeans project. When it is loaded, the name-value pairs are available for use in your application.

Properties files enable you to easily inject configuration information or other application data into the application.

Loading Properties from the Command Line

- Property information can also be passed on the command line.
- Use the –D option to pass key-value pairs:

```
java -Dpropertyname=value -Dpropertyname=value myApp
```

- For example, pass one of the previous values:

```
java -Dusername=user myApp
```

- Get the `Properties` data from the `System` object:

```
String userName = System.getProperty("username");
```


Property information can also be passed on the command line. The advantage to passing properties from the command line is simplicity. You do not have to open a file and read from it. However, if you have more than a few parameters, a properties file is preferable.

Resource Bundle

- The `ResourceBundle` class isolates locale-specific data:
  - Returns key/value pairs stored separately
  - Can be a class or a `.properties` file
- Steps to use:
  - Create bundle files for each locale.
  - Call a specific locale from your application.

Design for localization begins by designing the application so that all the text, sounds, and images can be replaced at run time with the appropriate elements for the region and culture desired. Resource bundles contain key-value pairs that can be hard-coded within a class or located in a `.properties` file.

Resource Bundle File

- Properties file contains a set of key-value pairs.
  - Each key identifies a specific application component.
  - Special file names use language and country codes.
- Default for sample application:
  - Menu converted into resource bundle

```
MessageBundle.properties
menu1 = Set to English
menu2 = Set to French
menu3 = Set to Chinese
menu4 = Set to Russian
menu5 = Show the Date
menu6 = Show me the money!
menuq = Enter q to quit
```


The slide shows a sample resource bundle file for this application. Each menu option has been converted into a name/value pair. This is the default file for the application. For alternative languages, a special naming convention is used:

```
MessageBundle_xx_YY.properties
```

where `xx` is the language code and `YY` is the country code.

Sample Resource Bundle Files

Samples for French and Chinese

MessagesBundle_fr_FR.properties

```
menu1 = Régler à l'anglais
menu2 = Régler au français
menu3 = Réglez chinoise
menu4 = Définir pour la Russie
menu5 = Afficher la date
menu6 = Montrez-moi l'argent!
menuq = Saisissez q pour quitter
```

MessagesBundle_zh_CN.properties

```
menu1 = 设置为英语
menu2 = 设置为法语
menu3 = 设置为中文
menu4 = 设置到俄罗斯
menu5 = 显示日期
menu6 = 显示我的钱!
menuq = 输入q退出
```


The slide shows the resource bundle files for French and Chinese. Note that the file names include both language and country. The English menu item text has been replaced with French and Chinese alternatives.

Initializing the Sample Application

```
PrintWriter pw = new PrintWriter(System.out, true);
    // More init code here

    Locale usLocale = Locale.US;
    Locale frLocale = Locale.FRANCE;
    Locale zhLocale = new Locale("zh", "CN");
    Locale ruLocale = new Locale("ru", "RU");
    Locale currentLocale = Locale.getDefault();

    ResourceBundle messages = ResourceBundle.getBundle("MessagesBundle",
     currentLocale);

    // more init code here

    public static void main(String[] args){
        SampleApp ui = new SampleApp();
        ui.run();
    }
```


With the resource bundles created, you simply need to load the bundles into the application. The source code in the slide shows the steps. First, create a `Locale` object that specifies the language and country. Then load the resource bundle by specifying the base file name for the bundle and the current `Locale`.

Note that there are a couple of ways to define a `Locale`. The `Locale` class includes default constants for some countries. If a constant is not available, you can use the language code with the country code to define the location. Finally, you can use the `getDefault()` method to get the default location.

Sample Application: Main Loop

```
public void run(){
    String line = "";
    while (!(line.equals("q"))){
        this.printMenu();
        try { line = this.br.readLine(); }
        catch (Exception e){ e.printStackTrace(); }

        switch (line){
            case "1": setEnglish(); break;
            case "2": setFrench(); break;
            case "3": setChinese(); break;
            case "4": setRussian(); break;
            case "5": showDate(); break;
            case "6": showMoney(); break;
        }
    }
}
```


For this application, a run method contains the main loop. The loop runs until the letter “q” is typed in as input. A string switch is used to perform an operation based on the number entered. A simple call is made to each method to make locale changes and display a formatted output.

The `printMenu` Method

Instead of text, a resource bundle is used.

- `messages` is a resource bundle.
- A key is used to retrieve each menu item.
- Language is selected based on the `Locale` setting.

```
public void printMenu(){
    pw.println("=== Localization App ===");
    pw.println("1. " + messages.getString("menu1"));
    pw.println("2. " + messages.getString("menu2"));
    pw.println("3. " + messages.getString("menu3"));
    pw.println("4. " + messages.getString("menu4"));
    pw.println("5. " + messages.getString("menu5"));
    pw.println("6. " + messages.getString("menu6"));
    pw.println("q. " + messages.getString("menuq"));
    System.out.print(messages.getString("menucommand")+" ");
}
```


Instead of printing text, the resource bundle (`messages`) is called and the current `Locale` determines what language is presented to the user.

Changing the `Locale`

To change the `Locale`:

- Set `currentLocale` to the desired language.
- Reload the bundle by using the current locale.

```
public void setFrench(){
    currentLocale = frLocale;
    messages = ResourceBundle.getBundle("MessagesBundle",
    currentLocale);
}
```


After the menu bundle is updated with the correct locale, the interface text is output by using the currently selected language.

Sample Interface with French

After the French option is selected, the updated user interface looks like the following:

```
=== Localization App ===
1. Régler à l'anglais
2. Régler au français
3. Réglez chinoise
4. Définir pour la Russie
5. Afficher la date
6. Montrez-moi l'argent!
q. Saisissez q pour quitter
Entrez une commande:
```


The updated user interface is shown in the slide. The first and last lines of the application could be localized as well.

Format Date and Currency

- Numbers can be localized and displayed in their local format.
- Special format classes include:
  - `java.time.format.DateTimeFormatter`
  - `java.text.NumberFormat`
- Create objects using `Locale`.

Changing text is not the only available localization tool. Dates and numbers can also be formatted based on local standards.

Displaying Currency

- Format currency:
  - Get a currency instance from `NumberFormat`.
  - Pass the `Double` to the `format` method.

- Sample currency output:

```
1 000 000 руб. ru_RU
1 000 000,00 € fr_FR
￥1,000,000.00 zh_CN
£1,000,000.00 en_GB
```


Create a `NumberFormat` object by using the selected locale and get a formatted output.

Formatting Currency with NumberFormat

```
 1 package com.example.format;
 2
 3 import java.text.NumberFormat;
 4 import java.util.Locale;
 5
 6 public class NumberTest {
 7
 8   public static void main(String[] args) {
 9
10     Locale loc = Locale.UK;
11     NumberFormat nf = NumberFormat.getCurrencyInstance(loc);
12     double money = 1_000_000.00d;
13
14     System.out.println("Money: " + nf.format(money) + " in
    Locale: " + loc);
15   }
16 }
```

ORACLE

Set the location and a numeric value to be displayed. Then, set up a `NumberFormat` object with a specified location. Pass the `Double` to the `format` method to print the formatted currency.

Displaying Dates

- Format a date:
  - Get a `DateTimeFormatter` object based on the `Locale`.
  - From the LocalDateTime variable, call the `format` method passing the formatter.
- Sample dates:

```
20 juil. 2011 fr_FR
20.07.2011 ru_RU
```


Create a date format object by using the locale and the date is formatted for the selected locale.

Displaying Dates with `DateTimeFormatter`

```
 3 import java.time.LocalDateTime;
 4 import java.time.format.DateTimeFormatter;
 5 import java.time.format.FormatStyle;
 6 import java.util.Locale;
 7
 8 public class DateFormatTest {
 9   public static void main(String[] args) {
10
11     LocalDateTime today = LocalDateTime.now();
12     Locale loc = Locale.FRANCE;
13
14     DateTimeFormatter df =
15       DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)
16         .withLocale(loc);
17     System.out.println("Date: " + today.format(df)
18         + " Locale: " + loc.toString());
19   }
```

ORACLE

The setup of the `DateTimeFormatter` is a bit verbose, but fairly clear. A factory is used to specify a style and a locale. Then the formatter is passed to the `LocalDateTime` object's `format` method.

Format Styles

- `DateTimeFormatter` uses the `FormatStyle` enumeration to determine how the data is formatted.
- Enumeration values
  - SHORT: Is completely numeric, such as 12.13.52 or 3:30 pm
  - MEDIUM: Is longer, such as Jan 12, 1952
  - LONG: Is longer, such as January 12, 1952 or 3:30:32 pm
  - FULL: Is completely specified date or time, such as Tuesday, April 12, 1952 AD or 3:30:42 pm PST

The `DateTimeFormatter` object uses the `FormatStyle` enumeration to format date, time or date/time.

Note: At the time of this writing, `FULL` and `LONG` can only be used with date or time return values. Only `MEDIUM` or `SHORT` can be used with date/time objects. Using the wrong value may result in a runtime error. We have not yet determined whether this is a feature or a bug.

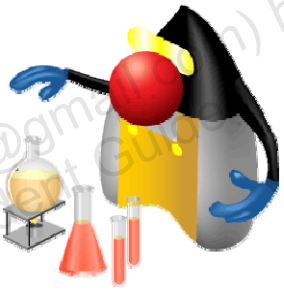
Summary

In this lesson, you should have learned how to:

- Describe the advantages of localizing an application
- Define what a locale represents
- Read and set the locale by using the `Locale` object
- Create and read a `Properties` file
- Build a resource bundle for each locale
- Call a resource bundle from an application
- Change the locale for a resource bundle

Practice 19-1 Overview: Creating a Localized Date Application

This practice covers creating a localized application that displays dates in a variety of formats.



Quiz

Which bundle file represents a language of Spanish and a country code of US?

a. `MessagesBundle_ES_US.properties`
b. `MessagesBundle_es_es.properties`
c. `MessagesBundle_es_US.properties`
d. `MessagesBundle_ES_us.properties`

Quiz

Which date format constant provides the most detailed information?

a. LONG
b. FULL
c. MAX
d. COMPLETE