

MULTILIZER™

G L O B A L L O C A L I Z A T I O N



Java Tutorial

Using MULTILIZER™ Java Edition 2.0

May 1999

Copyright © 1999 Innoview Data Technologies, Ltd. All rights reserved.

MULTILIZER is a trademark of Innoview Data Technologies, Ltd.
JBuilder is a registered trademark of Inprise Corporation.

Table of Contents

1	Preface	1
2	Introduction to the tutorial	2
3	Opening the Monolingual Application	3
4	Making the Application Multilingual	4
5	Creating a Dictionary for the Application	6
6	Internationalizing the Code.....	11
7	Changing Language On Run Time.....	16
8	Adding Western Languages	19
9	Adding Non-Western Languages	20
10	Writing Multilingual Applets	21
11	Writing Multilingual Swing Applications	22

1

Preface

The purpose of this tutorial is to familiarize you with common software localization tasks, when using MULTILIZER™. To obtain the most precise definitions on component use and technical details, please refer to the on-line help.

The following items can be found in **intro.pdf**

- Typographical conventions used in this document.
- General information on localization
- How does MULTILIZER™ work, what is Language Manager for?
- What is the "Native" language, used in Language Manager?

The following items can be found in **langman.pdf**

- Language Manager related tasks.

2

Introduction to the tutorial

In this tutorial we are going to create a multilingual application. The application will be a simple driving-time calculator, Dcalc which a user can use to calculate the average driving time for a given distance.

The Dcalc application is very simple but still it uses most of the features of MULTILIZER. The creation of the application is divided into several lessons each covering one or more MULTILIZER functions.

This tutorial is written for JBuilder 3. With each instructions there is also an explanation how to accomplish it using the plain JDK. If you use some other Java IDE (e.g. Visual Café, Visual Age, PowerJ, etc.) you can easily modify the procedure to match your Java IDE.



This symbol indicates that the information given applies to JBuilder only. In the front of a header it applies to the whole chapter, otherwise it applies to the current paragraph.



This symbol indicates that the information given applies to plain JDK only. In the front of a header it applies to the whole chapter, otherwise it applies to the current paragraph.



All screen-captured images have been taken when the active language of Language Manager is English. Set English on by clicking the Earth-image on the left side of the Language Manager's tool bar.

The Dcalc sample application locates in the `samples\dcalc` subdirectory of your MULTILIZER's Java directory.

To see more about how to use MULTILIZER read the online help and study the other sample applications found in the `samples` subdirectory.

If you used the Windows setup the MULTILIZER setup created the following program group:



Figure 1. Multilizer Program Group.

Before you can start building Dcalc you have to install MULTILIZER beans. To get the information on how to install them, see the readme files. Double click the *Java Readme* icon of the compiler to open the readme. If you installed MULTILIZER from a ZIP file read `index.html` file.

3

Opening the Monolingual Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. This is what we are going to do. The `samples\tutorial` contains the English Dcalc. Open it, compile it, and finally run it.

The application should look like this:

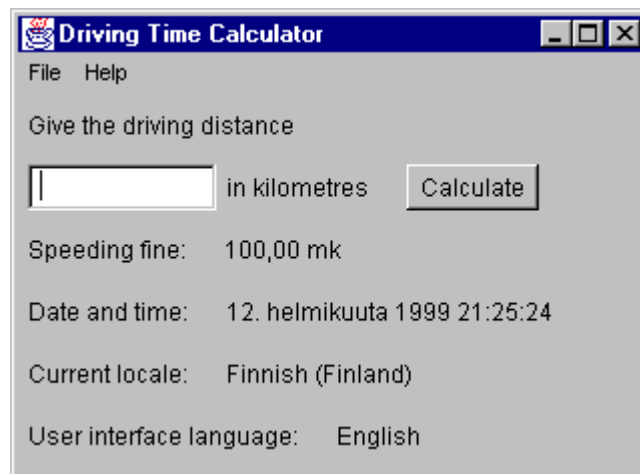


Figure 2. *The monolingual application using English.*

The user interface language is English and the application uses the default locale, which in this case is Finnish (Finland). The speeding ticket is formatted using the Finnish currency format (marks) and the date and time is also formatted using the Finnish format. Standard Java provides this kind of localization.

In the following chapters we will make Dcalc truly multilingual step-by-step.

4

Making the Application Multilingual



The first step is to make Dcalc multilingual, by just dropping two components on the form. Select the Translator component from the Component Palette. Drop the *multilizer.Translator* on the form. Drop the *multilizer.TestDictionary* component on the form as well.



The result should look like this:

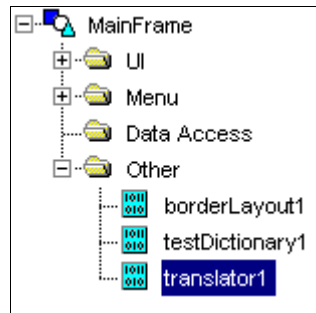


Figure 3. The translator and dictionary components have been added on the form.



Add the following code to the below of the import lines:

```
import multilizer.*;
```



Add the following code just before the constructor of MainForm:

```
TestDictionary testDictionary1 = new TestDictionary();
Translator translator1 = new Translator();
```

What are these two components for? TestDictionary is one of the dictionary components of MULTILIZER. A dictionary component provides string or phrase translation for the application. Normally each application contains one dictionary component that contains all the translation data of the application. MULTILIZER contains several different dictionary components: one for getting the translation data from a text file, an other for data from a database, etc.

The TestDictionary is a special case. It does not require any dictionary data but it makes the translation on the fly by mangling the original string to a test string. In a normal case you can not use the test dictionary in your final application because the translations are not a real language. However, the test dictionary is really handy in the development phase.

The Translator on the other hand is the component that does all the work. It scans the form before it becomes visible and translates the user interface string from the original value to the current language.



Select *translator1* component and move to the Properties windows. Drop down the value list of the *host* property. Select *this*. This specifies the control that the translator should translate. In the most cases it is the frame containing the translator component.



Add the following code to the *jbInit* function:

```
translator1.setHost(this);
```

Add the *translator1.translate()* line to the constructor of the MainForm:


```

public MainFrame()
{
    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
    try
    {
        jbInit();

        fineLabel.setText(NumberFormat.getCurrencyInstance(
            Locale.getDefault()).format(new Integer(100)));

        dateLabel.setText(DateFormat.getDateTimeInstance(
            DateFormat.LONG,
            DateFormat.MEDIUM,
            Locale.getDefault()).format(new Date()));

        localeLabel.setText(Locale.getDefault().getDisplayNames(Locale.UK));

        translator1.translate();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}

```

The translate method makes the translator to translate its host control. A proper place to call this is the last line of the constructor.

Compile and run Dcalc. It should look like this:

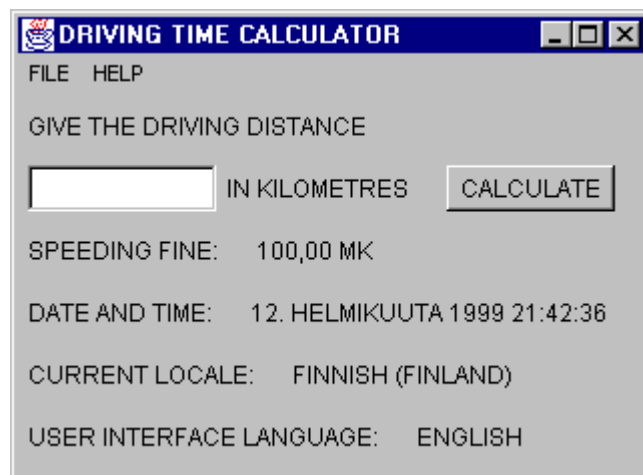


Figure 4. “Translated” application. The test dictionary translated every string to upper cased string.

As you can see, every user interface string is now in upper case. The translator changed every string type property after the form had been loaded from the resource. By default the test dictionary translates every string by putting it in upper-case.



For additional information on using the test dictionary, see the online help topic "TestDictionary".

This was a quick demonstration of the power of MULTILIZER. In the next chapter we will create a real dictionary that contains real languages.

5

Creating a Dictionary for the Application

Double click the Language Manager icon from the MULTILIZER program group to start Language Manager.



Language Manager is a Windows application. If your development environment is not Windows you have to manually create the dictionary. If you have Windows but you installed MULTILIZER from the platform independent ZIP you won't have Language Manager. In such case download Language Manager from MULTILIZER's web page and install Language Manager. For information on the dictionary file formats, find the online help topic "File Formats".

Choose File | New to start the Dictionary Wizard. The following dialog box appears:

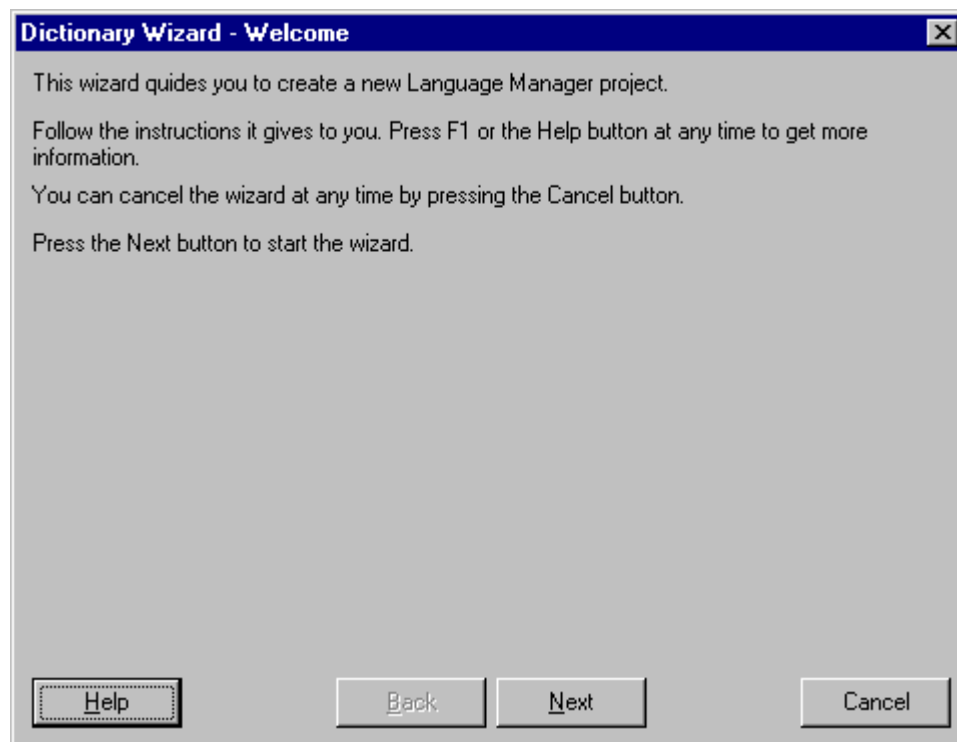


Figure 5. The Welcome sheet.

Press the Next button. The Source Directory sheet appears.

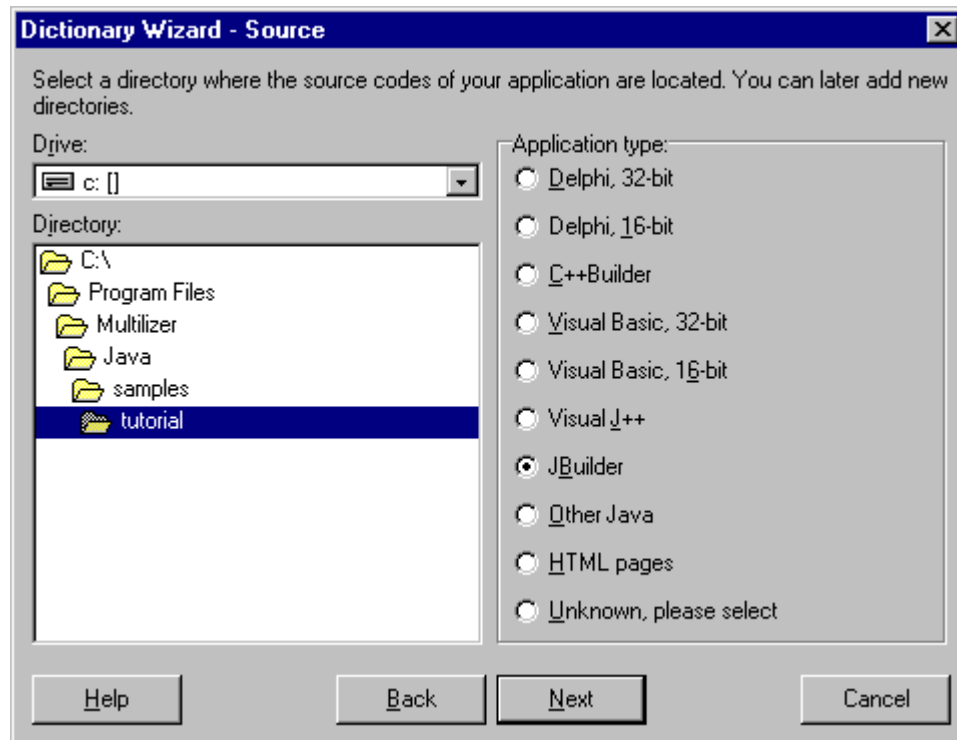


Figure 6. The Source sheet is used to enter the source directory.

This sheet specifies the directory where your application is located. Choose the `samples\tutorial` subdirectory of your MULTILIZER setup. If your application locates on multiple directories you can later add more directories. Dictionary Wizard detects the application type. If it is wrong you can select the right type.

Press the Next button. The Project Information sheet appears:

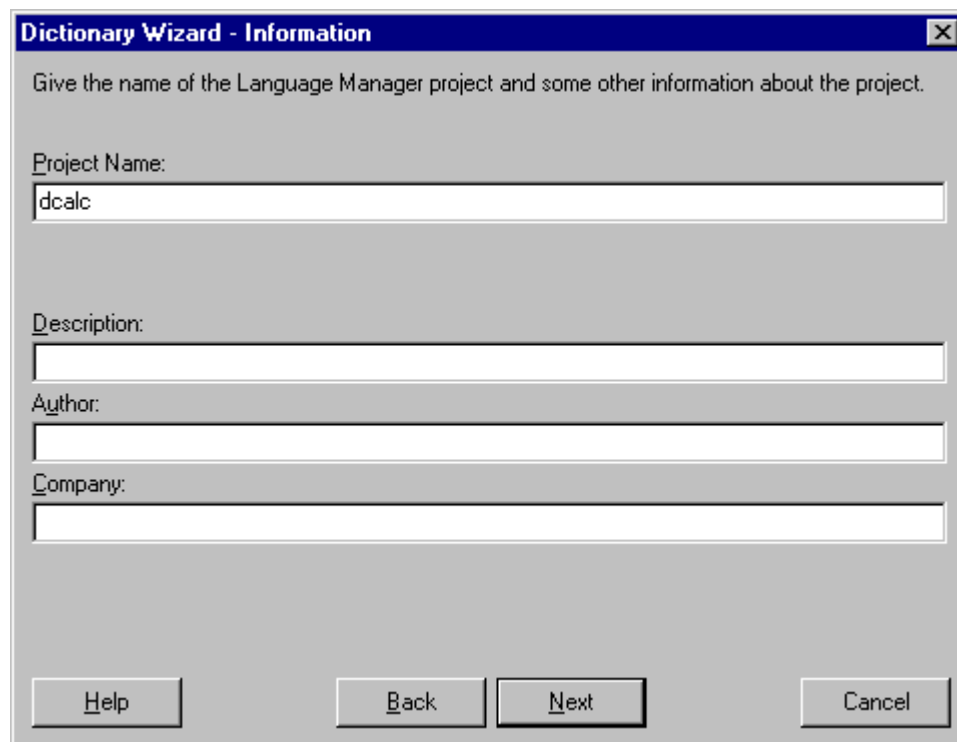


Figure 7. The Information sheet is used to enter the project name and application file.

This sheet specifies the directory name. You can also enter other information about the dictionary, the author, and the company.

Press the Next button. The Dictionary Type sheet appears:

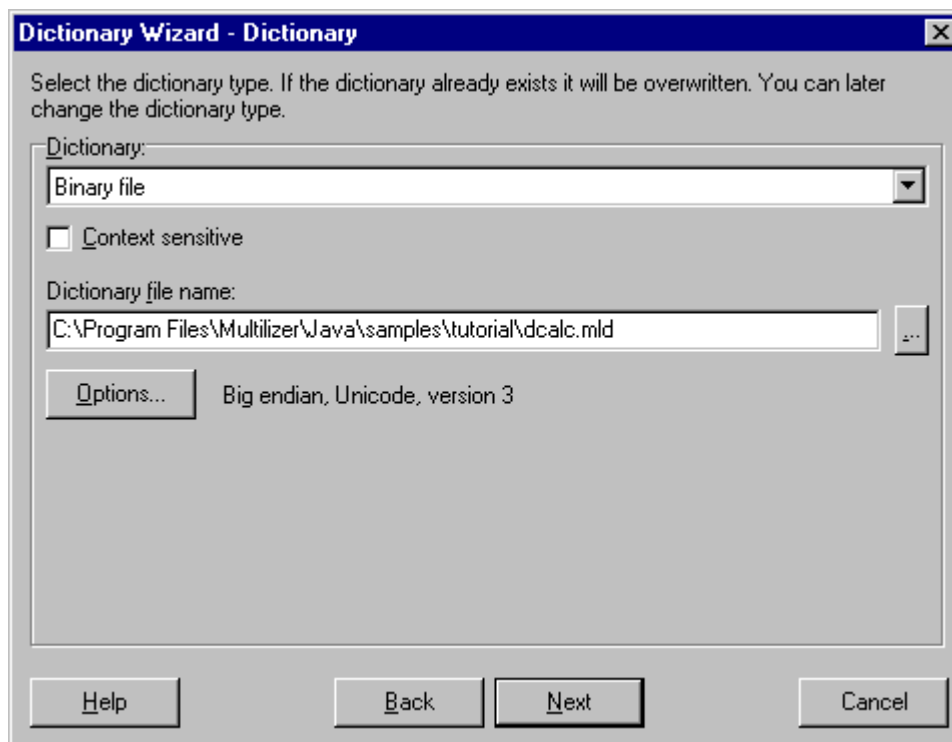


Figure 8. The Dictionary sheet is used to specify the dictionary type.

This sheet specifies the type of dictionary. The online help describes each dictionary type. Change the type from binary to text, accept the default values and press the **Next** button.

The Languages sheet appears:

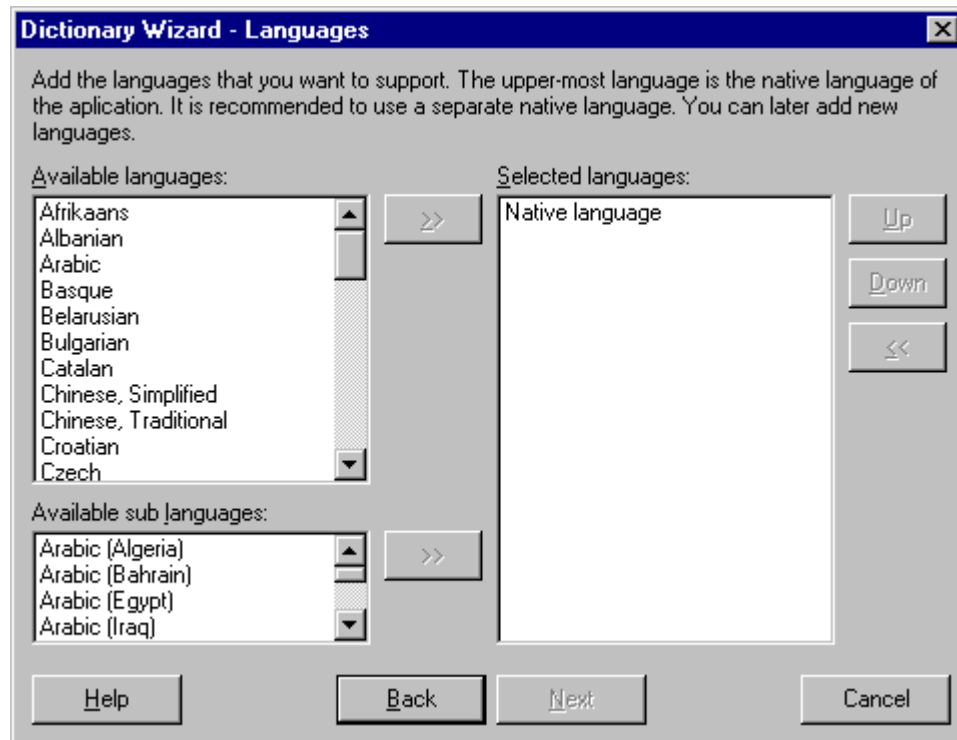


Figure 9. The Languages sheet is used to add languages to the dictionary.

This sheet lets you add string languages to the dictionary. By default the dictionary contains the native language. This is the language you used to program your application. From Available languages select English and press the >> button. This adds English support to the dictionary.

Add some other European language (we will take care of Far Eastern and Middle Eastern languages later). If you add Finnish the result should look like this:

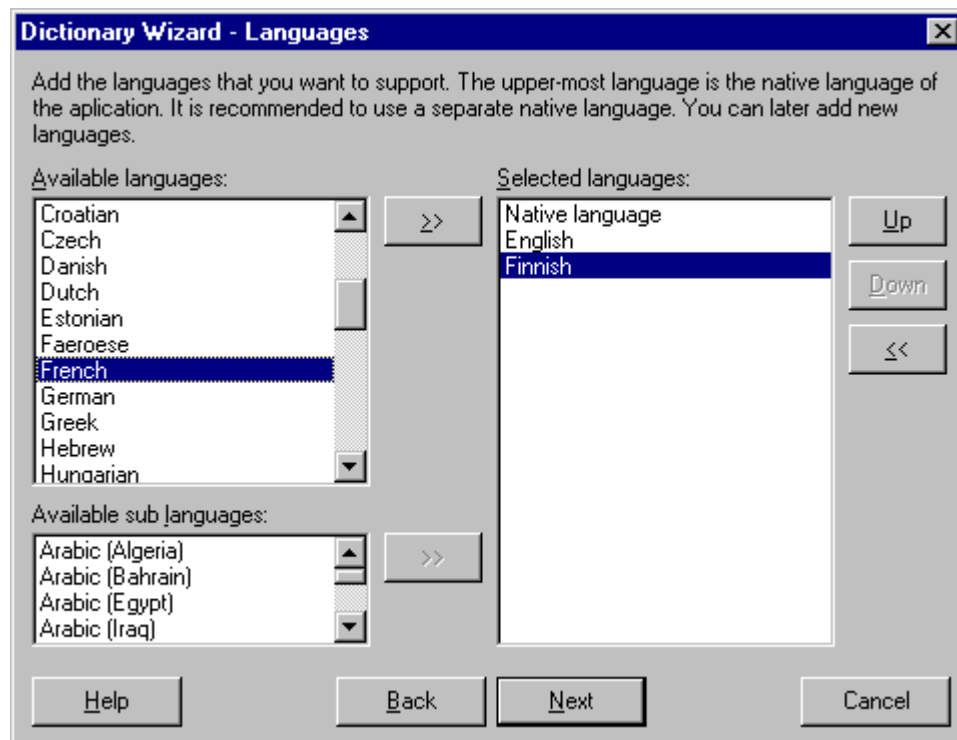


Figure 10. Native, English and Finnish added to dictionary.

Press the **Next** button. The Ready to create dictionary sheet appears. Now you have almost finished creating the dictionary.

Press the **Finish** button to end Dictionary Wizard. The following dictionary grid appears.

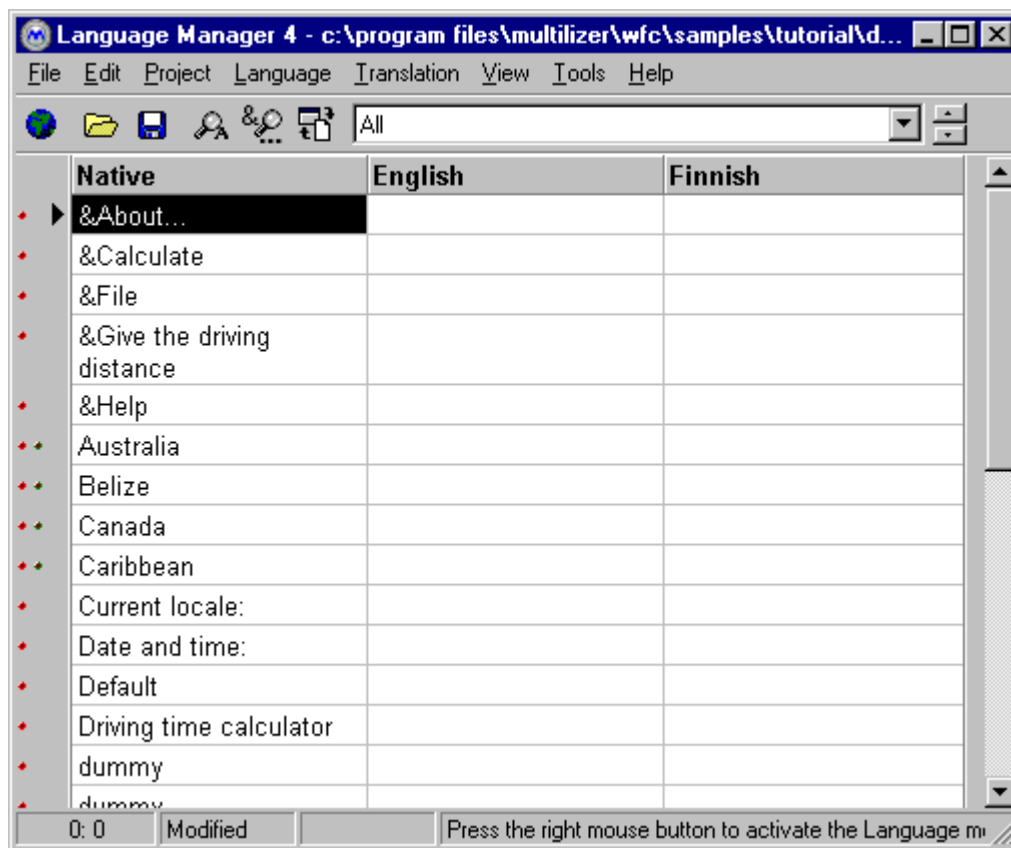


Figure 11. The dictionary grid.

Save the dictionary by choosing File | Save As.

We could translate the Finnish (or your own language) column manually by entering the translations. However there is an easier way: using the glossaries. They implement the translation memory. The glossary files contain the translations of the most common strings or phrases (e.g. File, Open, About, etc). To make your translation job easier we can first let Language Manager use the glossaries to translate those strings.

Language Manager contains several glossary files for most supported languages. You can edit the glossary files list by choosing Tools | Glossaries.



For additional information on glossaries, find the online help topic "Glossaries".

Open Dcalc's dictionary by choosing the uppermost item from the File | Reopen menu.

The next task is to translate the Finnish (or your own language) column. We can now use the master dictionary to translate some of the strings. Right mouse click the header of the Finnish column. A popup menu appears. Choose Translate | Using Glossaries. Language Manager translates most of the strings. It is up to you to translate the rest.

Use the arrow keys to move to the right cell and start typing. Translate every row. Because the native language is English you do not have to translate the English column. The dictionary uses the native string if the translation is not found.



For additional information on coping with failing translations, find the online help topic "MissingTranslation".

6

Internationalizing the Code



We have the dictionary files now. Let's use them. Delete the `TestDictionary` component from the form. Add the `multilizer.TextDictionary` component. Choose the component and move to the Properties window. Set the `fileName` property to `dictionary.languages` and the `translationFileName` property to `dictionary.translations` (i.e. the dictionary that you created in previous lesson). Set the `name` property to `dictionary1`.



The Properties window should look like this:



Figure 12. The Properties window showing the properties of a binary dictionary component.

Let's study some of the properties. The `language` property specifies the active language. By default it is `-1`. This makes MULTILIZER check the current locale of the user and find the language that matches the locale. If none is found the first (non-native) language is used.

The `locale` property specifies the active locale. The active language determines the language of the user interface. The active locale, however, determines the locale used by the application. The locale is a country and language specific object that controls how the date, time, currency, number, etc. are formatted.

In our case the dictionary contains English and Finnish. If the locale setting of the user is Finnish (Finland) the user interface of Dcalc will be in Finnish and the locale will be Finnish (Finland).



Add the following code just before the constructor of `MainFrame`:

```
TextDictionary dictionary1 = new TextDictionary();
```



Add the following code to the `jbInit` function.

```
dictionary1.setFileName("dictionary.languages");
dictionary1.setTranslationFileName("dictionary.translations");
```

Run the application. It should look like this:

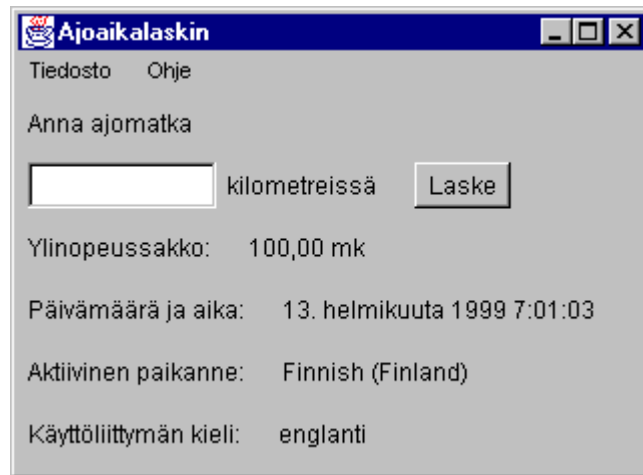


Figure 13. *Dcalc in Finnish.*

Making a multilingual application is this simple. In a simple case, this is all you have to do to make a multilingual application. In most other cases you have to do a little bit more.

If the program contains items which are just country (locale)-specific and hard coded in the source, they must be removed. This phase is called internationalization: it makes your software international and language/country independent. The next phase would then be to localize the program, i.e., add for each target country the locale-specific issues. This is done easily by using MULTILIZER. The remaining document discusses how to do this.

Dcalc calculates the average driving time. Most countries use the metric system, where the distance is expressed in kilometers. However in the US miles are used. Let's study how to make Dcalc compatible with both kilometers and miles.

When pressing the Calculate button Dcalc calls the following event:


```

void calculateButton_actionPerformed(ActionEvent e)
{
    int distance;

    try
    {
        distance = Integer.valueOf(textField.getText().trim()).intValue();
        if (distance < 0)
            throw new NumberFormatException();

        String[] params =
        {
            new Integer(distance/100).toString(),
            new Integer((int)(60*(distance%100)/100)).toString()
        };

        MessageDialog.messageBox(
            this,
            "Driving Time",
            MessageFormat.format("The average driving time is {0} hours and {1}
minutes", params),
            MessageDialog.OK);
    }
    catch (NumberFormatException ex)
    {
        String[] params = { textField.getText() };

        MessageDialog.messageBox(
            this,
            "Invalid value",
            MessageFormat.format("\">{0}\\" in not a valid distance", params),
            MessageDialog.OK);
        textField.requestFocus();
    }
}

```

When the English (United States) locale is on the user gives the distance in miles. To convert miles to kilometers add the following just before

```
String[] params =;
```

```

if (Utils.getMeasurementSystem(dictionary1.getActiveLocale()) ==
    Utils.US_MEASUREMENT)
    distance = (int)Utils.MILE_IN_METERS*distance/1000;

```

This is enough for to system to convert miles to kilometers but not for the user. He or she will most definitely be a bit confused if the user interface still prompts in kilometers. To make user interface react on the locale change adds the *languageChanged* event to the translator component and writes the following code:

```

void translator1_languageChanged(DictionaryEventObject e)
{
    if (Utils.getMeasurementSystem(dictionary1.getActiveLocale()) ==
Utils.US_MEASUREMENT)
        unitLabel.setText(translator1.translate("in miles")); //ivlm
    else
        unitLabel.setText(translator1.translate("in kilometres")); //ivlm

    fineLabel.setText(NumberFormat.getCurrencyInstance(
        dictionary1.getActiveLocale()).format(new Integer(100)));

    dateLabel.setText(DateFormat.getDateTimeInstance(
        DateFormat.LONG,
        DateFormat.MEDIUM,
        dictionary1.getActiveLocale()).format(new Date()));

    localeLabel.setText(Utils.getLocaleName(
        dictionary1.getActiveLocale(), dictionary1));
    languageLabel.setText(dictionary1.translate(
        dictionary1.getLanguageData().englishName));
}

```



Add the following code the the jblnit function. It adds the languageChanged event to the translator.

```

translator1.addLanguageChangeListener(new multilizer.DictionaryListener()
{
    public void languageChanged(DictionaryEventObject e)
    {
        translator1_languageChanged(e);
    }
});

```

First the code checks the measurement system. This is done by comparing the *measurementSystem* variable of the active locale. The code updates the text and help string. Let's study the following code in more detail:

```
unitLabel.setText(translator1.translate("in kilometres"));
```

In a monolingual application you would have used the following code:

```
unitLabel.setText("in kilometres");
```

This isn't the proper way in a multilingual application because the same EXE file must work on every language and locale. That's why the native string is translated before being assigned to the Caption property.

The lower part of the event updates the speeding fine, current time, active language name, and active locale name.

The constructor monolingual MainFrame contains the following code:

```

fineLabel.setText(NumberFormat.getCurrencyInstance(
    Locale.getDefault()).format(new Integer(100)));

dateLabel.setText(DateFormat.getDateTimeInstance(
    DateFormat.LONG,
    DateFormat.MEDIUM,
    Locale.getDefault()).format(new Date()));

localeLabel.setText(Locale.getDefault().getDisplayName(Locale.UK));

```

This code is not required any more because the *dictionary1_languageChanged* event updates the labels. You can remove it.

We need to make a few modifications to the *calculateButton_actionPerformed* event to make the message boxes multilingual. Consider the following code:

```
MessageDialog.showMessageDialog(  
    this,  
    "Driving Time",  
    MessageFormat.format("The average driving time is {0} hours and {1}  
minutes", params),  
    MessageDialog.OK);
```

MULTILIZER can not translate the standard message dialogs. You must use MULTILIZER's own *multilizer.MessageDialog* or add a translator component to the message dialog.

```
multilizer.MessageDialog.showMessageDialog(  
    this,  
    "Driving Time", //ivlm  
    Utils.formatMessage(  
        "The average driving time is {0} hours and {1} minutes", //ivlm  
        params,  
        dictionary1),  
    MessageDialog.OK,  
    new Translator(dictionary1));
```

Remember that you have to translate the exception message, as well.

```
multilizer.MessageDialog.showMessageDialog(  
    this,  
    "Invalid value", //ivlm  
    Utils.formatMessage(  
        "\"{0}\" in not a valid distance", //ivlm  
        params,  
        dictionary1),  
    MessageDialog.OK,  
    new Translator(dictionary1));
```

Translate the message box in the *aboutMenuItem_actionPerformed* event.

```
multilizer.MessageDialog.showMessageDialog(  
    this,  
    "About DCALC", //ivlm  
    "DCALC calculates the driving time.", //ivlm  
    MessageDialog.OK,  
    new Translator(dictionary1));
```

In this case you do not have to translate the text parameter but you can let the *MessageDialog* to translate it. This is because the message is not parametrized.

Most string constants in the above code examples are trailed with *ivlm* comment. The comment is a tag for Language Manager. The tag makes Language Manager to extract the string and to add it to the dictionary.

7

Changing Language On Run Time

Dcalc now has the ability to adapt to the current language and locale settings of the user. What about changing the language and/or locale on run time? Is this possible? Yes.



Double click the File menu and move to the white area (Type Here) on the bottom of the menu. Type **&Language....** Move the memo on the top of the *Exit* menu.



The result should look like this:

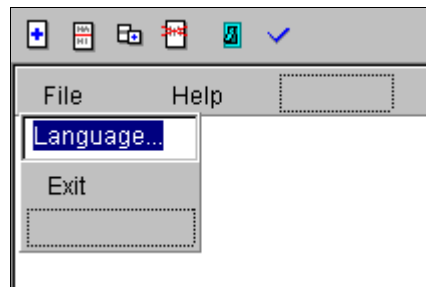


Figure 14. Add the Language menu item to the main menu.



Add the following code just before the constructor of MainFrame:

```
MenuItem menuItem1 = new MenuItem();
```



Add the following code into the jblnit function:

```
menuItem1.setLabel("Language...");
menuItem1.addActionListener(new java.awt.event.ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        menuItem1_actionPerformed(e);
    }
});
fileMenu.add(menuItem1);
```

Write the following event handler to the Language... menu item:

```
void menuItem1_actionPerformed(ActionEvent e)
{
    SelectLanguageDialog dialog = new SelectLanguageDialog(
        this,
        new Translator(dictionary1),
        false);

    if (dialog.showModal())
        dictionary1.setLanguage(dialog.getLanguage());
}
```

The dialog box contains a list of available languages that the user can select. After showing the dialog the event sets the new active language by setting the *language* property of the dictionary component.

Run the application and choose File | Language... (Or Tiedosto | Kieli... if you have Finnish active). The following dialog box appears:

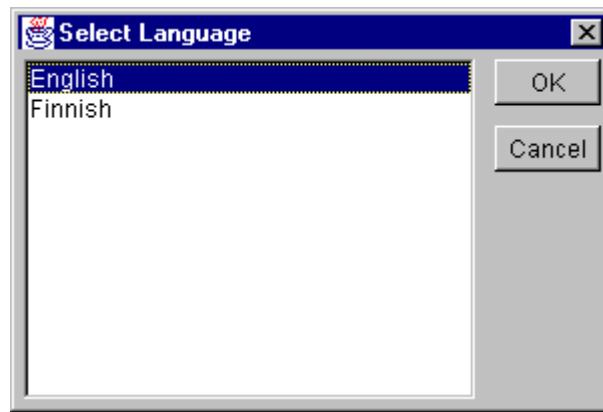


Figure 15. The *Select Language* dialog box lets the user select the active language on run time.

The *SelectLanguage* dialog box shows all available languages in a list box.

Select a new language and press the OK button. The active language (user interface) of Dcalc changes to that language. By default the active locale also changes to the default locale of the language. You can set the active language and locale independently by setting the *binding* property of the dictionary component to false.

The *SelectLanguageDialog* dialog box contains strings that need to be translated as well. Also the *MessageDialog* uses several strings (e.g. "OK", "Cancel"). The string tables of Language Manager contains all these constant strings. All you need to do is to add them to your dictionary. Launch Language Manager. Open the dictionary. **Choose Project | Include | System String**. The System String dialog box appears. Check Language Dialog, and Message Dialog check boxes.



Figure 16. The *System Strings* dialog box lets the developer add strings used by the system or by the standard components.

Press the OK button. Language Manager adds the strings used by the selected items. The glossaries contain translations of these strings. You do not have to translate them. Let Language Manager get the translations from the glossaries: Select any cell in the Finnish column. Choose **Language | Translate | Using Glossaries**.

Now our Dcalc application is fully multilingual. The user interface, locale settings, and input measures match the local settings. The user can even change the language on run time. The remaining chapters describe some of the advanced features of MULTILIZER.

8

Adding Western Languages

It is quite likely that you need to add new languages to your applications after the original languages. How to do that? This is the most powerful feature of MULTILIZER. After making your application multilingual, adding new languages is a piece of cake. You do not have to change the source code at all. Neither do you have to change the resources (forms) in any way. In fact, you do not even have to recompile the application.

Let's add Swedish to the dictionary. Start Language Manager. Choose **File | Open** and browse the `Java\Samples\Tutorial\dcalc.lmp`. Choose **Project | Languages**.

The Languages dialog box appears:

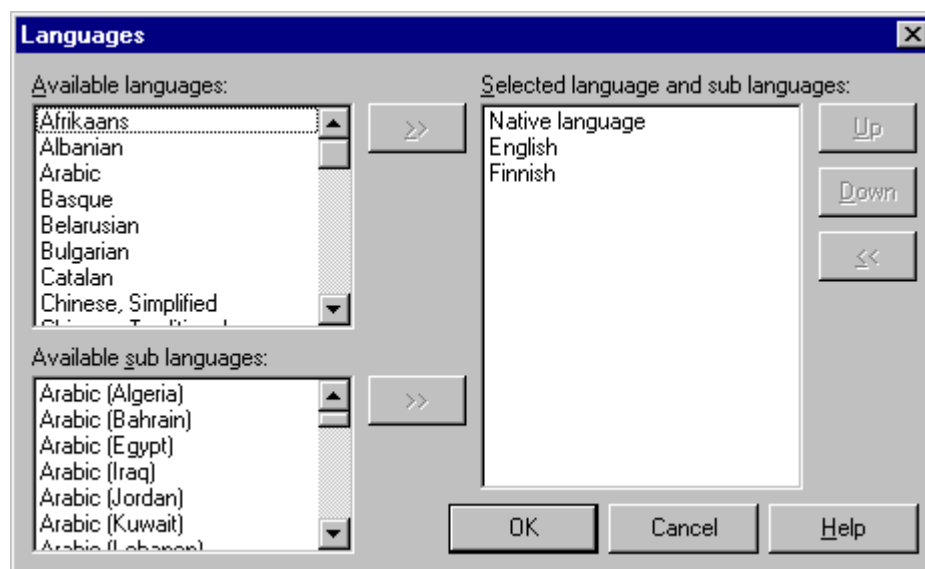


Figure 17. The Languages dialog box lets the user add or remove languages.

Select Swedish from the available languages and press the >> button. Press the **OK** button to close the dialog box. The Swedish column appears in the dictionary grid. Translate it and save the project. Next time you run Dcalc, Swedish is available.

9

Adding Non-Western Languages

Java uses Unicode strings. That's why in theory every Java application can display any characters. Unfortunately Java must always work on the top of the host platform (e.g. Windows, Linux, Solaris). The host platform does not necessary contain the font support needed by the language.

You might need to update the `font.properties` files of your Java runtime environment to add Far Eastern and Middle Eastern fonts.

10

Writing Multilingual Applets

Writing multilingual applets is as easy as writing multilingual applications. However you have to notice the following items:

- To make the applet compatible with most browser use the AppletTraslator component instead of the Translator component.
- Set the applet property of the dictionary component to refer to the applet.



For additional information on writing applets, see the Dcalc and Euro applets from the samples subdirectory.

11

Writing Multilingual Swing Applications

Writing multilingual Swing application is as easy as writing multilingual AWT applications. However you have to notice the following items:

- Add the `SwingModule` bean to the main frame. This adds the `Translator` bean the ability to translate the Swing components.

Learn more from the [online documentation](#).