Internationalization in Java

Advanced Programming

Internationalisation features of Java

- Amongst other things, Java is an Internet programming language.

- People who use the Internet may not necessarily speak or use English.

- What is useful to a speaker of English may be useless to a speaker of Spanish or Russian.

- Many web sites maintain different versions of their web site pages to cater to the different language communities that they service.

Internationalisation features of Java

- Java is the first computer language designed to provide direct support for internationalization.

- This means that it allows your programs to be customized for any number of languages in any number of countries without requiring cumbersome changes to the code.

Internationalisation features of Java

- The following are the major features of Java that support internationalization.

  – Java characters use Unicode.

  – Java provides the Locale class.

  – Java uses the ResourceBundle class for locale specific information

Unicode

Java characters use Unicode

- This is a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's languages.

- Using Unicode encoding makes it easier to write Java programs that manipulate strings in any international language.

Locales

Java provides the Locale class

- The Locale class encapsulates information about a specific locale.

- A locale determines how **locale-sensitive** information like date, time, and numbers is displayed and how locale-sensitive operations are performed.

- The classes for formatting date, time and numbers and for sorting strings are grouped in the java.text package.

Java Resource Bundles

- Java uses the ResourceBundle class for locale specific information such as status messages, exceptions and GUI component labels from the program.

- The information is stored outside the source code and can be accessed and loaded dynamically at runtime from a ResourceBundle, rather than hard-coded into the program.

Locales

- A Locale object represents a geographical, political, or cultural region in which a specific language or custom is used.

- For example, Irish, English, Americans all speak English. Chinese and Taiwanese speak Chinese. Russians speak Russian.

Locales

- The conventions for formatting dates, numbers, currencies, and percentages frequently differ from one country to another, e.g.:

    - The Chinese use year/month/day for representing date.

    - The Americans use month/day/year.

    - The Irish use day/month/year.

Locales

- One country alone does not necessarily define a locale.

- Canadians, for instance, speak Canadian English or Canadian French, depending which region of Canada you happen to be in.

Creating Locales

- To create a Locale object, you can use the following constructors in the java.util.Locale class.

  - Locale(String language, String country)
  - Locale(String language, String country, String variant)

- The language should be a valid language code.

- A valid language code is one of the lowercase, two letter codes defined by ISO-639. For example, **zh** stands for Chinese, **da** for Danish, **en** for English, **de** for German, **ko** for Korean.

Technical contents of ISO 639:1988 (E/F)
Code for the representation of names of languages

Two-letter lower-case symbols are used.

The Registration Authority for ISO 639 is Infoterm,
Österreichisches Normungsinstitut (ON), Postfach 130, A-1021
Wien, Austria.

Contains additions from ISO 639/RA Newsletter No. 1/1989,
and from a decision of the Advisory Committee of ISO/TC37 on
1997-08-27 in Copenhagen.

Allez à la version française.
Téir go dtí an leagan Gaeilge.

Code   English name of the language
 aa   Afar
 ab   Abkhazian
 af   Afrikaans
 am   Amharic
 ar   Arabic
 as   Assamese
 ay   Aymara
 az   Azerbaijani
 ba   Bashkir
 be   Byelorussian
 bg   Bulgarian
 bh   Bihari
 bi   Bislama
 bn   Bengali; Bangla
 bo   Tibetan
 br   Breton
 ca   Catalan
 co   Corsican
 cs   Czech

cy   Welsh
da   Danish
de   German
dz   Bhutani
el   Greek
en   English
eo   Esperanto
es   Spanish
et   Estonian
eu   Basque
fa   Persian
fi   Finnish
fj   Fiji
fo   Faroese
fr   French
fy   Frisian
ga   Irish (recte Irish Gaelic)
gd   Scots Gaelic (recte Scottish Gaelic)
gl   Galician
gn   Guarani
gu   Gujarati
gv   Manx Gaelic
ha   Hausa
he   Hebrew (formerly iw)
hi   Hindi

Creating Locales

- The country code should be a valid ISO country code, that is, an UPPERCASE, two-letter codes as defined by ISO-3166.

    – CA stands for Canada
    – CN for China
    – DK for Denmark
    – DE for Germany

Creating Locales

- The third argument (variant) is rarely used and are needed only for exceptional or system dependent situations to designate information specific to a browser or vendor.

- For example, the Norwegian language has two sets of spelling rules, the traditional one called Bokmål, and a new one called Nynorsk.

- The locale for traditional spelling is created as follows:

  - Locale("no", "NO", "B");

Using Locales

- Locale objects are really only of use if they are used in a particular context, e.g., currency display, or displaying a date/time

- Java provides the following "Locale sensitive" classes, i.e., the way they are displayed depends on the region:

  - Date (for dates)
  - Calendar (for months, day names)
  - Collator
  - DateFormat (for format of dates, e.g., US versus UK)
  - NumberFormat (for number formats, e.g. 3,000 versus 3.000 [UK versus EU]

```java
public class InternationalNumbers {
  public static void main(String[] args) {
      int interestRate=10,years=10;
      int loan=5,monthlyPay=2000,totalPay=60000;
      Locale locale = new Locale("de","DE"); //For Germany
      // Get formatters
      NumberFormat percForm = NumberFormat.getPercentInstance(locale);
      NumberFormat currencyForm = NumberFormat.getCurrencyInstance(locale);
      NumberFormat numberForm = NumberFormat.getNumberInstance(locale);
      percForm.setMinimumFractionDigits(2);
      // Display formatted input
      System.out.println(percForm.format(interestRate*12));
      System.out.println(numberForm.format(years));
      System.out.println(currencyForm.format(loan));
      // Display results in currency format
      System.out.println(currencyForm.format(monthlyPay));
      System.out.println(currencyForm.format(totalPay));
      //Some internationalized date stuff!!!
      DateFormatSymbols dfs = new DateFormatSymbols(locale);
      String dayNames[] = dfs.getWeekdays();
      // Set calendar days
      for (int i=0; i<7; i++)
       System.out.println(dayNames[i+1]);
  }
}
```

Change locale and EVERTHING changes

What appears on the screen depends on locale setting

Getting the Locales available

- All locale-sensitive classes contain a static method getAvailableLocale( ) which returns an array of the locales they support, for example:

  - Locale[ ] availableLocales = Calendar.getAvailableLocales( );

- This returns all locales for which Calendars are installed.

Resource Bundles

What are resources?

- In the general sense a resource is a supply which can be used in order to achieve an end, e.g. the natural resources of a country are used to supply energy for electricity etc.

- In computer programming a resource could be any of the following: a file containing message strings, buttons labels, label text; and image or sound file; files with binary data etc.

- Suppose you defined all of the strings in a java program in an external file...then it would be possible to change the display of your program merely by editing the resource file and without needing to recode

- This type of separation is very useful when building internationalized java programs

java.util.ResourceBundle

- Java has the means to define an object representing such a resource

- The class java.util.ResourceBundle represents a resource which can be accessed by a java program

- The resource can be any valid type, i.e. any object which extends from the 'cosmic superclass' java.lang.Object (which incidentally covers ALL objects)...use the *getObject()* method to retrieve arbitrary resource types

- The ResourceBundle supplies a method called *getBundle()* which retrieves a resource e.g.

  ResourceBundle res = res.getBundle(String bundleName,Locale loc)

- As you can see the *getBundle()* method also accepts a Locale so that the appropriate localized resource can be found

Creating ResourceBundle

- In general each resource is Locale specific e.g. a German resource file is written in German

- In Java to create a ResourceBundle for a particular Locale create a class and extend the ResourceBundle base class e.g.

```
public class ProgramResource extends ResourceBundle {

  private String[] keys = {"computeButton"};


  public Enumeration getKeys() {
     return Collections.enumeration(Arrays.asList(keys));
  }
  public Object handleGetObject(String key) {    return keys; }

}
```
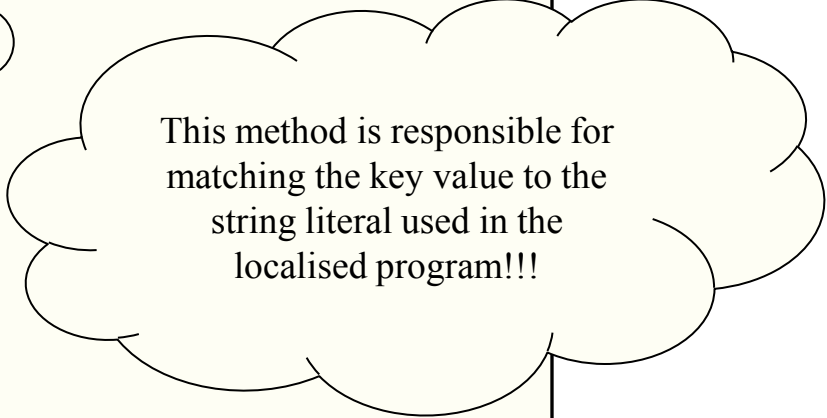
Creating ResourceBundle

- In the previous slide a ResourceBundle was declared but no Locale specific identifiers were added to the class name..this class is the top of the resource hierarchy for our all our locales, it determines that the keys shall be returned as a list

- This class also defines the **key values** that are used in your program to identify a particular localized resource e.g. a button which has "Compute" written on it for English speakers and "Rechnen" for German speakers, the key value for the button is "computeButton"

- In order to write a locale specific class first decide on the name of your subclass (using the ISO language and country codes) and then extend the base class of your ResourceBundle hierarchy

- For example the German language resource has the name

ProgramResource_de

Creating ResourceBundle's

```
//German language resource bundle
public class ProgramResource_de extends ProgramResource {


 public Object handleGetObject(String key) {


  String returnString="";


  if(key.equals("computeButton")) {
   returnString = "Rechnen";
  }
  else if(key.equals("greeting")) {
   returnString = "Wilkommen";
  }
  return returnString;
 }
}
```

This method is responsible for matching the key value to the string literal used in the localised program!!!

Naming the resource classes

- Follow the ISO standards for naming of resources

- The general format is:

  ProgramResource_language_country_variant

- As with Locales you do not need to supply all three codes every time you define a new locale, any of the following forms are acceptable:

  ProgramResource_language_country_variant
  ProgramResource_language_country
  ProgramResource_language
  ProgramResource

Naming the resource classes

- There's one overwhelmingly good reason for naming your classes this way the following example explains why

- Consider that the locale specific resource on the previous slide is the only resource class we have written, i.e. **ProgramResource_de**

- The following lines of code create a locale for 'de' and 'DE' and then define a resource based on the same Locale object:

  //Define a locale for german and Germany
  Locale loc = new Locale("de", "DE");

  //Create a resource bundle for German language
  ResoureBundle res = res.getBundle("ProgramResource",loc);

Loading the resource classes

- The java environment will search for a ResourceBundle class based on the name supplied in the first parameter combined with the Locale codes e.g. 'de' and 'DE'

- Once this class has been found it will be loaded into the runtime environment for execution....but there is no ProgramResource_de_DE class available to the class loader, so this class will fail to load....!!!!!

- But this program won't stop yet!!!....the *getBundle()* method will continue to match classes which are higher up in the hierarchy but dropping the _DE from the class identifier....and therefore attempting to load a class called ProgramResource_de...this class does exist and therefore we get the closest match to our desired locale!!!!

- This class matching will continue even if the _de did not match, and so the ProgramResource class would be loaded...this functionality is very useful

Loading the resource classes

- In general the resource lookup will occur according the following steps:

  1. Search for the class with the exact locale extension

  2. If no such class exists drop the variant, if specified, and attempt to load

  3. If no such class exists drop the country code, if specified, and attempt to load

  4. If no such class exists drop the language code, if specified, and attempt to load

  5. If still not found, use the default locale

  6. If no default locale found throw a MissingResourceException

Another Resource

- This class fits into the same resource hierarchy as before but deals with the English language and US country

```java
public class ProgramResource_en_US extends ProgramResource {

  public Object handleGetObject(String key) {

    String returnString="";

    if(key.equals("computeButton")) {
      returnString = "Compute";
    }
    else if(key.equals("greeting")) {
      returnString = "Welcome";
    }
    return returnString;
  }
}
```

Using the ResourceBundle's

- Once each locale has it's own class definition your java program can be written without any string literals used e.g. suppose you wanted to say "Welcome" in your program, normally you'd define a String with value "Welcome"

- With ResourceBundle's defined, instead of having String literals (or other objects which are locale-sensitive) in your program you define a key value, e.g. "greeting"

- This key value is a locale-sensitive identifier and then the actual String can be retrieved via the ResourceBundle method e.g. *res.getString("message"); //*Where res is a ResourceBundle reference

- In the current example *getString("message")* will return the String "Wilkommen" for the _de Locale and "Welcome" for english Locales

- *getObject(String)* is used when the resource is not only a String

```java
//Sample GUI that uses resource bundles to internationalize
public class SimpleButtonGUI extends JFrame {

 ResourceBundle res;

 public SimpleButtonGUI() {

  Locale loc = new Locale("de","DE"); //create Locale for German in Germany

  res = res.getBundle("ProgramResource",loc); //Create resource bundle

  JLabel greetLabel = new JLabel(res.getString("greeting")); //get value for greeting key value

  JButton computeButton = new JButton(res.getString("computeButton")); //button key value

  getContentPane().add(greetLabel);
  getContentPane().add(computeButton,BorderLayout.SOUTH);

  setSize(200,200);
  setVisible(true);
 }

 public static void main(String[] args) {
  SimpleButtonGUI myGui = new SimpleButtonGUI();
 }
}
```

Adding extra locales

- Adding extra locales to this sample program is straight forward...create new subclasses of the base class ProgramResource using the naming convention described earlier

- Each of the new locale specific subclasses will implement the *handleGetObject(String)* method but the if statement will return the locale specific String's e.g. "Bienvenue" for French key value "greeting"

- However, there are considerable overhead's in creating all of these if statements for every string in your program....

- What can be done to reduce these overheads???....you can use the **ListResourceBundle** or **PropertyResourceBundle** classes!!!

java.util.ListResourceBundle

- ListResourceBundle is particularly useful when the resources being localized are not strings

- Instead of the ResourceBundle classes extending the ResourceBundle class the ListResourceBundle class is extended e.g.

```
public class ProgramResource_de extends ListResourceBundle {


 private static final Object[][] contents = { {"computeButton","Rechnen"},
{"greeting","Wilkommen"} };


 public Object[][] getContents() { return contents; }

}
```

java.util.ListResourceBundle

- This subclass contains a two-dimensional array of Object's which is a list of the key value pairs in the locale

- The *getContents()* returns the Object array contents to the ListResourceBundle base class which carries out the processing needed to return the correct resource based on the key input (it calls the *getObject()* method in the background)

- Using the ListResourceBundle will simplify the addition of new locale-sensitive resources to the program

- Using the ListResourceBundle class requires no change to the GUI shown to demonstrate the ResourceBundle

java.util.PropertyResourceBundle

- If the java program being created has a lot of String literals which are locale-sensitive, then using the PropertyResourceBundle will simplify the localization of these strings

- The PropertyResourceBundle reads an <u>external property file</u> which contains ONE KEY VALUE PAIR PER LINE

- A property file is simply a text file used to define properties e.g.

  ```
  computeButton=Rechnen

  greeting=Wilkommen
  ```

- This property file could have the name "PropertyFile_de.txt", for example, indicating that it localizes to German

java.util.PropertyResourceBundle

- In order to read information from the property file, an InputStream object must be instantiated

- This InputStream object is then passed to the PropertyResourceBundle which will use this to retrieve the correct literal for the key value passed

- Fortunately java makes this easy by supplying a method *getResourceAsStream* method in the PropertyResourceBundle class which you can access using class reflection in your resource subclass e.g.

  ProgramProperties_de.class.getResourceAsStream("ProgramProperties_de.txt")

- Passing this to the superclass constructor will set up the ResourceBundle to read the property file

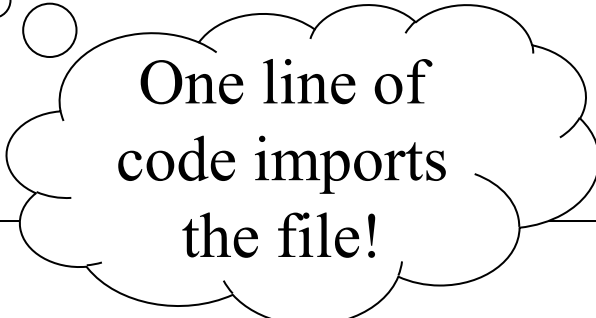# Using PropertyResourceBundle

```
//A PropertyResourceBundle for German


public class ProgramProperties_de extends PropertyResourceBundle {


  public ProgramProperties_de() throws IOException {

    super(ProgramProperties_de.class.getResourceAsStream("ProgramPro

    perties_de.txt"));

  }

}
```

One line of code imports the file!

```java
public class SimplePropFileGUI extends JFrame {

 ResourceBundle res;

 public SimplePropFileGUI() {
   Locale loc = new Locale("de","");
   res = res.getBundle("ProgramProperties",loc);


   JLabel greetLabel = new JLabel(res.getString("greeting"));
   JButton computeButton = new JButton(res.getString("computeButton"));
   getContentPane().add(greetLabel);
   getContentPane().add(computeButton,BorderLayout.SOUTH);
   setSize(200,200);
   setVisible(true);
 }


 public static void main(String[] args) {
   SimplePropFileGUI myGui = new SimplePropFileGUI();
 }

}
```
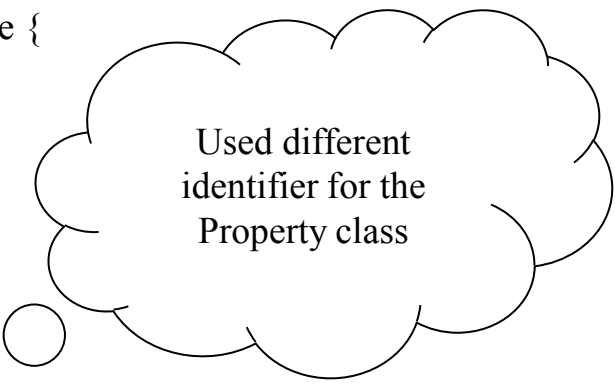
Used different identifier for the Property class

ResourceBundle summary

- Use the PropertyResourceBundle class for localization of strings...this means that the translator doesn't even have to know how to edit Java code!!!

- Use the ListResourceBundle class when non-String objects need to be localized

- ResourceBundle's are extremely useful, especially when combined with Locale objects

Playing Sound in Java
Applet Sound Engine makes playing sounds in Java simple…

Java Sound engine

- You can use the Java Sound engine in standard JDK to play audio data from both applications and applets.

- Java Sound enables you to play many types of audio clips, including AIFF, AU, WAV, MIDI, and RMF files.

- This overview describes the features provided by the Java Sound engine and how it is integrated with standard JDK and the other Java Media technologies.

- It shows you how you can play consistent, reliable, high-quality audio from your Java programs.

Java Sound engine

- Since JDK 1.2 enables you can create and play AudioClips from both applets and applications.

- The clips can be any of the following audio file formats:

    AIFF
    AU
    WAV
    MIDI (type 0 and type 1 files)
    RMF

- The sound engine can handle 8- and 16-bit audio data at virtually any sample rate.

- If the hardware doesn't support 16-bit data or stereo playback, 8-bit or mono audio is output.

Playing Sounds

- With the standard JDK you can play many different types of audio files from both applets and applications, unlike JMF that needs to be specifically added/installed

  - Audio is stored in files. There are several formats of audio files.

  - The JDK can play several audio file formats, including .wav and .au files.

Playing Sounds

- *play(URL url, String filename);*

  Plays the audio clip after it is given the URL and the file name that is relative to the URL. Nothing happens if the audio file cannot be found.

- *play(getCodeBase(), "soundfile.au");*

  Plays the sound file soundfile.au, located in the applet's directory.

- *play(getDocumentBase(), "soundfile.au");*

  Plays the sound file soundfile.au, located in the HTML file's directory.

Using AudioClip interface

- *public AudioClip getAudioClip(URL url);*

- *public AudioClip getAudioClip(URL url, String name);*

  Either method creates an audio clip. Specify String name to use a relative URL address.

  public abstract void play()
  public abstract void loop()
  public abstract void stop()

  Use these methods to start the clip, play it repeatedly, and stop the clip, respectively.

Playing sound in Applets

To play a sound file, load the clip by using Applet.getAudioClip and control playback through the AudioClip play, loop, and stop methods.

- *For example, to play a WAV file from an applet, you could*

1. *Call Applet.getAudioClip and pass in the URL where the .wav file is located.*

2. *Call play or loop on the AudioClip.*

- *The audio data is loaded when the AudioClip is constructed.*

Code to play a sound

Regardless of the type of sound file used, the code for loading and playing the file is the same.

This example provides a framework for loading and playing multiple audio clips and loads the audio clips asynchronously.

The code for loading and playing the clips is:

AudioClip audioClip = Applet.getAudioClip(baseURL, relativeURL);
onceClip.play();   //Play it once.
loopClip.loop();   //Start the sound loop.
loopClip.stop();   //Stop the sound loop.

Play a sound in an Application

Applications as well as applets can play sounds.

The newAudioClip method is contained in the java.applet.Applet class to enable applications to create AudioClips from a URL.

public static final AudioClip newAudioClip(URL r)

Play a sound in an Application

To play a sound from an applet, you call Applet.newAudioClip to load the sound and then use the AudioClip play, loop, and stop methods to control playback.

To play a WAV file from an application, you could therefore ...

1. *Call Applet.newAudioClip and pass in the URL where the .wav file is located.*

2. *Call play or loop on the AudioClip.*

Play a sound in an Application

A sound player in the previous example can easily be implemented as an application.

The main difference is that Applet.newAudioClip is called to load the sounds.

> *AudioClip audioClip = Applet.newAudioClip(completeURL);*

Testing the sound application

- *We have seen that Java programs can manipulate and play audio clips.*

- *You can therefore capture your own clips and play them in your programs.*

- *Obviously you need the proper equipment to achieve this*

  … sound card in the PC and speakers or headphones.

AudioClip for simple sound types

The sound engine that plays the audio-clips supports several audio file formats:

Sun Audio file format
⇒ **.au extension**

Windows Wave file format
⇒**.wav extension**

Macintosh AIFF file format
⇒ **.aif or .aiff extension**

Musical Instrument Digital Interface (MIDI) file format
⇒**.mid or .rmi extensions**

To play more complex sounds you may have to use the Java Media Framework supports many additional formats.

Labwork Exercise 1

- Create a new project in Eclipse

- Create a JFrame application that contains <u>ONE</u> JButton with the label "List All Locales" and <u>ONE</u> JTextArea. Output ALL available Locales to the JTextArea when the JButton is pushed.

Labwork Exercise 2

- Create a new project in Eclipse

- Modify the JFrame from the previous slide. Add a JComboBox with "English" and "French" listed . Use Resource bundles to translate the english JButton to French "Montrer tous les Locales" when the French option is selected. The JComboBox should also be internationalized to read "Francais" and "Anglais" when French is the selected language.

Labwork Exercise 3

Create a JFrame in Eclipse with one button to play a sound file (wav or au) of your choice using the **AudioClip** interface