11-localization-and-zones

Contents

Preface **1**
Locale . . . 2
Structure . . . 2
Details . . . 3
Creating . . . 4
Resource bundles . . . 4
Usage . . . 5
Loading . . . 8

Summary **8**

- Preface
  - Locale
    - Structure
    - Details
    - Creating
  - Resource bundles
    - Usage
    - Loading
- Summary

Preface

Computers and software have become so prevalent today that they are used everywhere in the world, for human activities. For any software to be relevant and useful to these users, it needs to be localized. The process in which we adapt the software to the local language and customs is known as localization.

`Localization` is all about making the software relevant and usable for the users from different cultures - in other words, customizing software for people from different countries or languages. How is that achieved ? Two important guidelines should be heeded when you localize a software application

- Do `not hardcode` text (such as messages to the users, textual elements in the GUI etc.). and separate them into external files or dedicated classes. With this accomplished there is usually minimal effort to add support for a new local in the software.
- Handle `cultural specific aspects` such as date, time and currency, formatting with localization in mind. Instead of assuming a default locale, design in such a way that the current locale is fetched and customized.

Locale

A locale is place representing a country, language or culture. Consider the Canada-French locale, French is spoken in many parts of Canada, and this could be a locale. In other words, if one wants to sell software that is customized for Canadians who speak French then one needs to facilitate the software for this locale. In Java, this locale is represented by the code fr_CA, where fr is short for French (the language) and CA is short for Canada (for the country code).

Method	Description
static Locale[] getAvailableLocales()	Returns a list of available locales (i.e., installed locales) supported by the JVM.
static Locale getDefault()	Returns the default locale of the JVM.
static void setDefault(Locale newLocale)	Sets the default locale of the JVM.
String getCountry()	Returns the country code for the locale object.
String getDisplayCountry()	Returns the country name for the locale object.
String getLanguage()	Returns the language code for the locale object.
String getDisplayLanguage()	Returns the language name for the locale object.
String getVariant()	Returns the variant code for the locale object.
String getDisplayVariant()	Returns the name of the variant code for the locale object.
String toString()	Returns a String composed of the codes for the locale's language, country, variant, etc.

```
System.out.println("The default locale is: " + Locale.getDefault());
Locale [] locales = Locale.getAvailableLocales();
System.out.printf("No. of other available locales is: %d, and they are: %n",
   locales.length);
Arrays.stream(locales).forEach(locale -> System.out.printf("Locale code: %s
   and it stands for %s %n", locale, locale.getDisplayName()));
```

Structure

And the output of the snippet above, might look something like that. Note that the output is abridged to only show the general format in which the locales are printed and their respective display names

```
The default locale is: en_US
No. of other available locales is: 160, and they are:
Locale code: ms_MY and it stands for Malay (Malaysia)
Locale code: ar_QA and it stands for Arabic (Qatar)
Locale code: is_IS and it stands for Icelandic (Iceland)
Locale code: sr_RS_#Latn and it stands for Serbian (Latin,Serbia)
Locale code: no_NO_NY and it stands for Norwegian (Norway,Nynorsk)
Locale code: th_TH_TH_#u-nu-thai and it stands for Thai (Thailand,TH)
Locale code: fr_FR and it stands for French (France)
Locale code: tr and it stands for Turkish
Locale code: es_CO and it stands for Spanish (Colombia)
Locale code: en_PH and it stands for English (Philippines)
Locale code: et_EE and it stands for Estonian (Estonia)
Locale code: el_CY and it stands for Greek (Cyprus)
Locale code: hu and it stands for Hungarian
```

```
[... rest of the output is elided ... ]
```

There are multiple types of of localization formats

- "hu" - stands for Hungarian, just one code where the code is the language code, doubles as country code and variant too
- "ms_MY" - stands for Malay (Malaysia) - language and country code
- "no_NO_NY" - stands for Norwegian (Norway, Nynorsk) - language, country and region codes
- "th*TH_TH*#u-nu-thai" - two or three initial codes separated with, language, country code, variant and the extension is `u-nu-thai`

Here is the general format of the locale or localization string. The locale coding scheme allows combining different variations to create a locale.

`The format of the locale: language + "_" + country + "_" + (variant + "_#" | "#")+ script + "-" + extensions, note that country code and variant are always written out in capital letters, the language is written out as lower case.`

Consider English which is spoken in many countries. There are variations in English based on the country in which the language is spoken. We all know that American English is different from British English, but there are many such versions. Below is a quick snippet that prints out all english language based localization formats.

```
Arrays.stream(Locale.getAvailableLocales())
    .filter(locale -> locale.getLanguage().equals("en"))
    .forEach(locale -> System.out.printf("Locale code: %s and it stands for %
       s %n", locale, locale.getDisplayName()));
```

The output might look something like that, note that there are quite a few types of locales which are based off of english. The output refers to different locales in English and makes use of only language code and the country code.

```
Locale code: en_MT and it stands for English (Malta)
Locale code: en_GB and it stands for English (United Kingdom)
Locale code: en_CA and it stands for English (Canada)
Locale code: en_US and it stands for English (United States)
Locale code: en_ZA and it stands for English (South Africa)
Locale code: en and it stands for English
Locale code: en_SG and it stands for English (Singapore)
Locale code: en_IE and it stands for English (Ireland)
Locale code: en_IN and it stands for English (India)
Locale code: en_AU and it stands for English (Australia)
Locale code: en_NZ and it stands for English (New Zealand)
Locale code: en_PH and it stands for English (Philippines)
```

Details

The locale class has a few other methods, to obtain information about the language, country, variant and script extensions, which can all be part of the local string.

```
Locale.setDefault(Locale.CANADA_FRENCH);
Locale defaultLocale = Locale.getDefault();
System.out.printf("The default locale is %s %n", defaultLocale);
System.out.printf("The default language code is %s and the name is %s %n",
    defaultLocale.getLanguage(), defaultLocale.getDisplayLanguage());
```

```
System.out.printf("The default country code is %s and the name is %s %n",
    defaultLocale.getCountry(), defaultLocale.getDisplayCountry());
System.out.printf("The default variant code is %s and the name is %s %n",
    defaultLocale.getVariant(), defaultLocale.getDisplayVariant());
```

The snippet above will print out something along the lines of the following. Note that the variant code is empty so is the display name of the variant code. What is the difference between the methods such as `getCountry` and `getDisplayCountry`. The former returns the country code and the latter returns the human readable name of the country. The country code is a two or three letter code based on ISO 3166. The `getLanguage` and `getDisplayLanguage` is similar to the country code, it is based on the ISO 639. There was no variant in the locale, as mentioned, so nothing got printed out. However for some other locales there could be variant values and those values would get printed for that locale. The variant could be any extra details such as operating environments (like MAC for Macintosh machine) or name of the company (such as Sun or Oracle). Other than these one can also have less widely used methods such as the `getScript()` - which returns the script part code of the locale string (if a script is present in the extra field format)

```
The default locale is fr_CA
The default language code is fr and the name is fran ais
The default country code is CA and the name is Canada
The default variant code is and the name is
```

`Instead of calling Local's getDisplayCountry, method which takes no arguments, you can choose to overloaded version of the getDisplayCountry(Locale), which takes a Locale object as an argument. This will print the name of the country as in the passed locale. For example, for the call Locale.GERMANY.getDisplayCountry()wil print out "Deutschalnd" (that is how germans refer to their country)however for the call Locale.GERMANY.getDisplayCountry (Locale.ENGLISH)the output will be Germany, that is the British refer to the country Germany.`

Creating

There are many ways to get or create a Locale object. Below are listed four options here for creating an instance of Italian locale, that corresponds to the language code of it.

```
Locale locale1 = new Locale("it", "", "");
Locale locale2 = new Locale("it");
Locale locale3 = new Locale.Builder().setLanguageTag("it").buld();
Locale locale4 = Locale.ITALIAN;
```

The way the locale is created is surely based on needs, if the language thag is coming from user input, one should use the constructors, otherwise the predefined constants can be used instead.

Resource bundles

The resource bundles are tightly related to the Locale, they are meant to hold resources based on the Locale type. They are used to customize the behavior of the program based on the locale. One obvious solution is to get the default locale, check if the locale is Italy, and print “Ciao”, while it will work, but this approach is neither flexible nor extensible. How about customizing to other locales like Saudi Arabia or Thailand. One has to find and replace all the locale specific strings for customizing each locale. This task will be a nightmare if the application is designed like that, consisting of thousands of strings to be replaced, spread over a million lines of code, and there are many locales to support

In Java, resource bundles provide a solution to this problem of how to customize the application to local-specific needs. So what is a resource bundle ? A resource bundle is a set of classes or property files that

help define a set of keys and map those keys to locale specific values. The abstract class `ResourceBundle`, provides an abstraction of resource bundles in Java. It has two derived classes `PropertyResourceBundle` and `ListResourceBundle`. The two derived classes provide support for resource bundles using two different mechanisms:

- `PropertyResourceBundle` this concrete class provides support for multiple locales in the form of property files, For each locale, one needs to specify the keys and values in a property file, for that locale. For a given locale, if you use the `ResourceBundle.getBundle()` method the relevant property file will be automatically loaded. Of course there is no magic i it; there are certain naming conventions that need to be followed for creating the property files, which will be discussed below.
- `ListResourceBundle` - for adding support to a locale, extend this class. In the derived class the method `getContents` has to be overridden. Which returns and `Object[][]`. This array must have the list of keys and values, The keys must be Strings. Typically the values are also Strings, but values can be anything: sound clips, video clips, URLs or pictures.

Usage

As already mentioned if the application is designed with localization in mind in the first place, support for new locales can be added without doing any changes to the code. As discussed above, property files define strings as key value pairs in a file. Here is an example of a classpath that can be mapped to an actual path in your machine: `classpath=C:\Program File\Java\jre8`. Property files will usually contain numerous such key value pairs, with each such pair in separate lines, as in the following

```
classpath=C:\Program Files\Java\jre8
temp=C:\Windows\Temp
windir=C:\Windows
```

In the case of localization one uses property files to map the same key strings to different value strings. In the program, the key strings are used, and by loading the matching property file for the locale the corresponding values for the keys will be fetched from the property files for use in the program. The naming of these property files is important and below is the content of these bundles. To keep this example simple there is only one key-value pair in these property files; in real-world programs, there could be a few hundred or even thousands of pairs present in each property file

ResourceBundle.properties

```
Greeting=Hello
```

ResourceBundle_ar.properties

```
Greeting=As-salamu Alayjum
```

ResourceBundle_it.properties

```
Greeting=Ciao
```

As demonstrated above, the default bundle is named `ResourceBundle`.properties. The resource bundle for Arabic is named `ResourceBundle_ar`. Note that the suffix "_ar" indicating Arabic is the localized language. Similarly the resource bundle for Italian is named `ResourceBundle_it`, which makes use of the "_it" suffix to indicate the Italian as the associated language with this property file.

```
Locale currentLocale = Locale.getDefault();
ResourceBundle resBundle = ResourceBundle.getBundle("ResourceBundle",
   currentLocale);
System.out.printf(resBundle.getString("Greeting"));
```

The example above will load by default the default resource bundle which is the file `ResourceBundle.properties`. However if the locale is different, say it is manually set to Italy with the `Locale.setDefault(Locale.ITALY)`. Then the file that will be loaded is the properties file that matches the Italy locale, however if none is found, it will still default to the default one mentioned above. Manually setting the locale is not recommended since that is done by the `JVM`, based on the system settings, or can be done when starting the application with `java -jar -Duser.language=it -Duser.region=IT LocalizedHello`.

```
D:\> java LocalizedHello
Hello

D:\> java -Duser.language=it LocalizedHello
Ciao

D:\> java -Duser.language=ar LocalizedHello
As-Salamu Alaykum
```

This demonstrates how an application can be started from the command line, with different locale, by default the locale is assume to be en_US, (US English). The corresponding property file is loaded and the message string is resolved. Also note that the snippet above does not specify any suffixes for the name of the `ResourceBundle`, this is already mentioned, that the name of the file has to follow a specific convention, the name is always `ResourceBundle`, and the suffix is the same / matches as the format string of the locale

`ListResourceBundle`, is used when a support for new locale is required, by extending this class. While extending the `ListResourceBundle` class one needs to override the abstract method `getContents`, the signature of the method is - `protected Object[][] getContents()`. Note that the keys have to be String, but the values can be of any type, hence the array of type Object, further the method returns a list of keys and value pairs. As a result the getContents method returns a two dimensional array of Objects

`Resource bundles are created by extending the ListResourceBundle class, whereas with PropertyResourceBundle, the resource bundle is created as property file. Furthermore, when extending ListResourceBundle, as a value one can specify any type of object, whereas in the PropertyResourceBundle the value is restricted to be of type String only, the same as the key type`

```
// default US English version
public class ResBundle extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
    { "MovieName", "Avatar" },
    { "GrossRevenue", (Long) 2782275172L }, // in US dollars
    { "Year", (Integer)2009 }
    };
}

// Italian version
public class ResBundle_it_IT extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
    { "MovieName", "Che Bella Giornata" },
```

```
    { "GrossRevenue", (Long) 43000000L }, // in euros
    { "Year", (Integer)2011 }
    };
}

public class LocalizedBoxOfficeHits {
    public void printMovieDetails(ResourceBundle resBundle) {
        String movieName = resBundle.getString("MovieName");
        Long revenue = (Long)(resBundle.getObject("GrossRevenue"));
        Integer year = (Integer) resBundle.getObject("Year");
        System.out.println("Movie " + movieName + "(" + year + ")" + "
           grossed "
            + revenue );
    }

    public static void main(String args[]) {
        LocalizedBoxOfficeHits localizedHits = new LocalizedBoxOfficeHits();
        // print the largest box-office hit movie for default (US) locale
        Locale locale = Locale.getDefault();
        localizedHits.printMovieDetails(ResourceBundle.getBundle("ResBundle",
            locale));
        // print the largest box-office hit movie for Italian locale
        locale = new Locale("it", "IT", "");
        localizedHits.printMovieDetails(ResourceBundle.getBundle("ResBundle",
            locale));
    }
}
```

First thing that should be noted here, is that the resource bundle classes are not directly instantiated. They are actually loaded dynamically once the `getBundle` method is called, this happens during the run-time of the program using reflection internally the java implementation is creating instances of the classes that extend the `ListResourceBundle`. They are instantiated based on the required locale. These instances are then cached for further use, since more than likely the locale will be kept constant across the execution of the program, however that is `not always guaranteed`, that is why it is not really advisable to cache the current `ResourceBundle` object in user space, it is usually recommended to simply keep calling `getBundle` every time the resource bundle is required, or wrap that call in a static utility method, but do not cache the instance of `ResourceBundle` itself, since the locale might change.

```
// This code change will result in throwing this exception:
// Exception in thread "main" java.lang.ClassCastException:
// java.lang.Long cannot be cast to java.lang.Integer
Integer revenue = (Integer)(resBundle.getObject("GrossRevenue"));
```

When using ListResourceBundle, one needs to be extra careful about the type of the resource behind a given string key, since it is of type Object, it can be anything, in this case the GrossRevenue is strictly defined as Long, therefore it can not be cast to Integer just like that, one needs to use the method toInt()from the Long's interface to convert it to proper int first. This cast here is not trying to cast primitives, rather it is trying to cast class instances of incompatible types

```
// This code will crash with this exception:
// Exception in the thread "main" java.util.MissingResourceException:
```

```
// Can't find resources for bundle ResBundle, key GrossRevenu, due to typo
Long revenue = (Long)(resBundle.getObject("GrossRevenu"));
```

The key name in the ResourceBundle is important, meaning that it must match exactly, it is also case sensitive, otherwise an exception will be thrown, in this case the MissingResourceException, this is relevant for both PropertyResourceBundle and ListResourceBund resource bundle types

Loading

The process of finding a matching resource bundle is same for classes extended from `ListResourceBundle` as it is for property files defined for `PropertyResourceBundle`. For the resource bundles implemented as classes extended from `ListResourceBundle`, Java uses the reflection mechanism to find and load the class. The class has to be defined as public, and should be visible, so that the reflection mechanism will find the class.

Naming convention for the Resource bundles - both files and classes (extending from `ListResourceBundle`) are enforced by the language itself, Java provides naming convention to be followed for creating resource bundles. Only through the names of the property bundles does the Java library load the relevant locales. Hence it is important to understand and follow this naming convention when creating the property bundles for localizing Java applications.

```
packagequalifier.bundlename + "_" + language + "_" + country + "_" + (variant + "_#" |
"#")+ script + "-" + extensions
```

Given that there could be many resource bundles for a bundle name, what is the search sequence to determine the resource bundle to be loaded ? To clarify, the sequence is a series of steps. The search starts from the first step, and if at any of the steps a match is found the resource bundle is loaded. Otherwise the search process continues to the next step.

1. The search starts by looking for an exact match for the resource bundle with the full name.
2. The last component (separated by _) is dropped and the search is repeated with the resulting shorter name. This process is repeated until the last locale modifier is left e.g. - `ResourceBundle_th_TH_TH_#u-nu-thai` -> `ResourceBundle_th_TH_TH` -> `ResourceBundle_th_TH` -> `ResourceBundle_th`
3. Search for the resource bundle for the default locale (assume that the default configured locale is en_US) - `ResourceBundle_en_US` -> `ResourceBundle_en`
4. Search for the resource bundle with just the name of the bundle. - `ResourceBundle`
5. The search fails throwing a `MissingBundleException`

What happens is that the search is exhausting all possible names for a bundle, this is also very useful since this guarantees that as long as there is a default bundle, or a larger scope bundle it will be found, even if there is none for the current locale, guaranteeing that applications will still continue to work just fine, in the absence of some specific or niche locale for the current user or system. The serach starts by first looking at the most specific bundle name, based on the locale, gradually broadening the scope, by looking for a more generic bundle name

Summary

Read and set the locale

- A locale represents a language culture or country; the Locale class in Java provides an abstraction for this concept.
- Each locale can have three entries: the language and country to form locale tags. There are no standard tags for variants;
- The `getter` methods in the Locale class - such as `getLanguage` or `getCountry`, `getVariant` return codes; whereas the similar methods of `getDisplayName`, `getDisplayLanguage` and so on, return human readable text representation for those codes
- The `getDefault` method in Locale returns the default locale set in the `JVM`. This default locale can be changed to another locale by using the `setDefault` method.
- There are many ways to create or get a Locale object - using a `constructor`, `forLanguageTag`, `Locale .Builder` or from the predefined constants

Create and use properties file

- A resource bundle is a set of classes or property files that help define a set of keys and map those keys to locale-specific values.
- The class `ResourceBundle` has two derived classes `PropertyResourceBundle` and `ListResourceBundle` .
- To obtain the resource bundle use the `ResourceBundle.getBundle` method to get the bundle for a given locale, or the default one
- The `PropertyResourceBundle` class provides support for multiple locales in the form of property files, for each locale one needs to specify the keys and values in a property file located on the class path. Both the keys and values in the file are of type String
- To add support for a new locale extend the `ListResourceBundle`, class in the derived class override the abstract method `Object[][] getContents` The returned array must have the list of keys and values. The keys must be string and values can be of any type
- When passing the key string to the `getObject()` method to fetch the matching value in the resource bundle make sure that h passed keys and the key in the resource bundle exactly match (the key name is case sensitive). I the y do not match a `MissingResourceException` will be thrown
- The naming convention for a fully qualified resource bundle name is the `packagequalifier + the local format string`

Build resource bundle

- The process of finding a matching resource bundle is same for classes extended from `ListResourceBundle` as for property files defined for `PropertyResourceBundle`.
- The special search sequence to look for a matching resource bundle. Search starts from the most narrow - using the current locale, to the most broad - default `ResourceBundle`
- The `getBundle()` method takes a `ResourceBundle.Control` object as an additional parameter. By extending this `ResourceBundle.Control` class and passing that object one can control or customize the resource bundle searching and loading process