Advanced Programming

Internationalization

Localization

i18n & l10n

- **Internationalization** (i18n) is the process of designing an application so that it can be adapted to various languages and regions without engineering changes.
  - Most applications should be designed for international use right from the start of the development.
- **Localization** (l10n) is the process of adapting software for a specific region or language by adding locale-specific components and translating text.
  - Country-specific data should be added without changing the code!

Considerations

- With the addition of localized data, **the same executable can run worldwide** → adding support for new languages should not require recompilation.
- **Textual elements**, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Culturally-dependent data, such as **dates**, **numbers** and **currencies**, appear in formats that conform to the end user's region and language.
- Sorting collections of strings should take into consideration **the alphabet** specific to a language.
- The direction of writing, etc.

The *Locale* Class

- A *Locale* object represents a specific geographical, political, or cultural region.
  - **Language**: ISO 639 alpha-2 or alpha-3 language code

    “en” → english, “ja” → japanese, “ro” → română
  - **Country**: ISO 3166 alpha-2 country code

    “US” → United States, “UK” → United Kingdom, “RO” → Romania
  -
```
Locale usLocale = Locale.US;
Locale roLocale = new Locale("ro", "RO");
```
- An operation that requires a *Locale* to perform its task is called *locale-sensitive*
  - `NumberFormat.getNumberInstance(roLocale).format(12345.99)`

    → `12.345,99`

Available Locales

```
import java.util.Locale;

public class TestLocale {
  public static void main(String args[]) {
    System.out.println("Default locale:");
    localeInfo(Locale.getDefault());

    System.out.println("Available locales:");
    Locale available[] =
        Locale.getAvailableLocales();

    for(Locale locale : available) {
      locale.getDisplayCountry() + "\t" +
      locale.getDisplayLanguage(locale));
    }
  }
}
```

Default locale:
English United States

Available locales:
Albania shqip
Algeria العربية
Argentina español
Australia English
Austria Deutsch
Bahrain العربية
Belarus беларускі
Belgium français
Belgium Nederlands
Bolivia español
Bulgaria български
Canada français
Canada English
Chile español
China 中文
…
Romania română
…

Where Are Locale Data Stored?

- **Resource bundles** contain locale-specific objects
  When your program needs a locale-specific resource, a String for example, your program can load it from the resource bundle that is appropriate for the current user's locale.

```
ResourceBundle resources =
    ResourceBundle.getBundle("MyResources", someLocale);
```

- Resource bundles belong to families whose members share a common base name, but whose names also have additional components that identify their locales.

  - `MyResources`
  - `MyResources_ro`
  - `MyResources_en_US`

- A resource bundle can be a **.class** or a **.properties** file

- **rt.jar** → sun.util.resources.
  LocaleData, LocaleNames, CurrencyNames, TimeZoneNames, ...

Properties

- A **.properties** file is a text file containing pairs of type: **key = value**

config.properties

```
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost/test
user = dba
password = sql
```

- The **Properties** class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream.

```
Properties props = new Properties();
props.load(new FileReader(path + "/database.properties"));
System.out.println(props.getProperty("driver"));
```

ResourceBundles

- MyResources_**ro**.properties

```
hello=Salut
bye=La revedere
welcome=Utilizatorul {0} s-a conectat la {1}
```

- MyResources_**fr**.properties

```
hello=Bonjour
bye=Au Revoir
welcome=Utilisateur {0} est connecté à {1}
```

- MyResources.properties (default)

```
hello=Hello
bye=Goodbye
welcome=User {0} logged in at {1}
```

Translating the User Interface

```
public static void displayMessages(Locale locale) {
  String baseName = "com.example.MyResources";
  ResourceBundle messages =
     ResourceBundle.getBundle(baseName, locale);
  System.out.println(messages.getString("hello"));
  String pattern = messages.getString("welcome");
  Object[] arguments = {"Duke", LocalDate.now()};
  String welcome = new MessageFormat(pattern).format(arguments);
  System.out.println(welcome);
  System.out.println(messages.getString("bye"));
}

public static void main(String args[]) throws IOException {
  displayMessages(Locale.forLanguageTag("ro"));
 //-> Salut, Utilizatorul Duke s-a conectat la 2016-05-03, La revedere
  displayMessages(Locale.getDefault());
 //-> Hello, User Duke logged in at 2016-05-03, Goodbye
}
```

ListResourceBundle

- Using a **class** instead of a **.properties** file

```
package com.example;
import java.util.ListResourceBundle;
public class MyResources_ro extends ListResourceBundle {

  private static final Object[][] contents = {
    {"hello", "Salut"},
    {"welcome", "Utilizatorul {0} s-a conectat la {1}",
    {"bye", "La revedere"}
  };

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

Formatting Numbers

- **NumberFormat** helps you to format and parse numbers for any locale.
- Your code can be completely independent of the locale conventions for decimal points or thousands-separators.

```
double number = 12345.99;
NumberFormat.getNumberInstance(Locale.US).format(number));
    → 12,345.99
NumberFormat.getNumberInstance(Locale.GERMANY).format(number));
    → 12.345,99
```

- Formatting currency

```
double salary = 1_000_000d;
Locale roLocale = Locale.forLanguageTag("ro-RO");
NumberFormat.getCurrencyInstance(roLocale).format(salary));
    → 1.000.000,00 LEI
```

Formatting Dates

- **java.time**.format.DateTimeFormatter

```
LocalDateTime today = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter
          .ofLocalizedDate(FormatStyle.FULL)
          .withLocale(roLocale);
System.out.println("Date: " + today.format(formatter));
   // → Date: 03 mai 2016
```

- Format Styles:
  - ➔ SHORT
  - ➔ MEDIUM
  - ➔ LONG
  - ➔ FULL

Comparing Strings

- The **Collator** class performs locale-sensitive String comparison.

```
String words[] =
    {"ramură", "rămurică", "rățuscă", "repede", "rîu"};
Arrays.sort(words);
System.out.println(Arrays.toString(words));
//[ramură, repede, rîu, rămurică, rățuscă]


Locale roLocale = Locale.forLanguageTag("ro");
Collator collator = Collator.getInstance(roLocale);
Arrays.sort(words, (w1,w2) -> collator.compare(w1, w2));
System.out.println(Arrays.toString(words));
//[ramură, rămurică, rățuscă, repede, rîu]
```

Internationalization Layer

- **Separate the** internationalization **code** from the rest of the application

  → **encapsulate** your i18n code
- You may want to use the default Java I18N features, or you may use another API.
- Every input from the user should be **parsed** according to the rules defined by the I18N layer
- Every output to the user should be **formatted** according to the rules defined by the I18N layer

Bibliography

- **Java Tutorial**
  https://docs.oracle.com/javase/tutorial/i18n/TOC.html