



## xTier™ Internationalization Service

The xTier™ Internationalization Service provides a cross-paradigm internationalization facility for Java and .NET.

### Key Benefits:

- Unified XML-based format for all internationalization parameters which facilitates translation and editing of internationalized data
- Support for "skins" which allows run-time configuration of the look and feel of the application.

## Internationalization Overview

Internationalization (often abbreviated as i18n since there are 18 letters between the first "i" and the last "n"), is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. In other words, internationalization is the process where application code is designed in such a way that it is completely independent of any cultural information (for instance, there are no hard coded text labels). Such information is stored in external data files and is loaded at runtime. There are several categories of cultural specific information:

**Messages and labels:** No strings used in the user interface should be hard wired in the code. These strings should be externalized to a resource file and translated to each language that the application will support. Consequently the program can display the appropriate strings (based on the desired language) without modification to the application code.

**Character classification:** For example, in English there are upper case and lower case characters. Other languages can have more classifications to consider, and in some languages the uppercase/lowercase classification does not apply.

**Numeric and currency formatting:** Currency symbols and number grouping differ in each country.

**Date and time formatting:** Day/month/year order varies for different regions. For example, in Europe day comes first, then month, followed by the year, but in the USA month comes first, then day, followed by the year.

**Collation (sorting order):** – For Roman alphabet letters their ASCII values are actually compared to determine their sorting order. However, for other code pages it may not be applicable and thus some special rules have to be applied to find the proper sorting order.

An internationalized program has the following characteristics:

- With the addition of localized data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.
- Cultural specific data appears in formats that conform to the end user's region and language.
- It can be localized quickly.

Localization (often abbreviated as I10n because there are 10 letters between the "l" and the "n"), is the process of adapting software for a specific region or language by adding locale-specific components and translating text. Usually, the most time-consuming portion of the localization phase is the translation of text. Other types of data, such as sounds and images, may require localization if they are culturally sensitive. Localizers also verify that the formatting of dates, numbers, and currencies conforms to local requirements.

## Internationalization in JAVA

It is helpful to illustrate i18n principles using a simple Java example.

```
public class HelloWorld {  
  
    static public void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

In the preceding code, it is clear that in order to internationalize this code, the message "Hello World" must be translated. Further, once translated, the application code must be changed and then recompiled to support the new language. Consider internationalized version of this example. The source code for the internationalized program follows. Notice that the text of the messages is not hard coded.

```
import java.util.*;  
  
public class HelloWorld {  
  
    public static void main(String[] args) {  
        String language = "en";  
        String country = "US";  
  
        if (args.length == 2) {  
            language = args[0];  
            country = args[1];  
        }  
    }  
}
```

```
Locale locale = new Locale(language, country);
ResourceBundle resources = ResourceBundle.getBundle("HelloWorld",
    locale);
System.out.println(resources.getString("hello"));
}
```

To compile and run the program the developer must define a property file corresponding to the given locale, i.e. language and country. For example, a property file for the default locale should be named "HelloWorld.properties" (as discussed in a following section, the file name is important), and should define a key-value pair for the "hello" property:

```
hello=Hello World!
```

In order to utilize this application in Germany, the "hello" property must be defined for the German language. The properties file should be named "HelloWorld\_de\_DE.properties", and should contain the following:

```
hello=Hallo Welt!
```

Let's examine how this works in detail. Notice in the example code that a string variable is passed as an argument to the **System.out.println()** and is no longer hard coded. Now, the appropriate value is loaded from the resource file at runtime. In order to retrieve the appropriate value, the local must be specified before retrieving the resources. (Please note that there must be a property file for every local that the application will support.) The locale identifies the particular language and country and is represented by the **java.util.Locale** class. Notice, the sample code constructs the locale object from command line arguments (or uses the default locale if language and country codes are not passed on the command line).

The next step is to obtain a **java.util.ResourceBundle** object corresponding to the given locale, by its name. **ResourceBundle** objects contain locale-specific objects and are used to isolate locale-sensitive data, such as translatable text messages. In the example above a **ResourceBundle** is obtained in the following way:

```
ResourceBundle resources = ResourceBundle.getBundle("HelloWorld", locale);
```

The **getBundle** method takes two arguments to identify the specific properties file to load. The first string argument ("HelloWorld" in this case) specifies the family of properties files. In this particular case, all filenames should begin with "HelloWorld":

```
HelloWorld.properties
HelloWorld_en_US.properties
HelloWorld_en_UK.properties
HelloWorld_de_DE.properties
```

The second argument is an instance of **Locale** that specifies exactly which file from the family of files to load. The country and language, which were passed to the **Locale** constructor, should correspond to the ones declared in the filename. For example, for a locale defined as:

```
Locale locale = new Locale("en", "UK");
```

the **HelloWorld\_en\_UK.properties** file will be chosen.

A properties file is a simple text file that consists of key-value pairs representing the localized properties. The key is used to retrieve the localized value does not depend on the locale and for each property remains constant in all localized property files and should be unique for a given property file. The value represents the localized property value for a given language and country. The value is divided from the key by the "=" sign. The localized value can be retrieved in the following way:

```
String localizedMessage = resources.getString("hello");
```

The argument "hello" passed to the **getString(String)** method is the property key, by which the value is retrieved from the appropriate property file.

## Locales

Let's examine the **Locale** object in detail. The **Locale** object can be created using a one or two parameter constructor:

```
Locale locale = new Locale("fr", "CA");  
Locale locale = new Locale("en", "GB");  
Locale locale = new Locale("en", "");  
Locale locale = new Locale("fr");
```

The first parameter is the language code which consists of two lower-case letters. The language code must conform to ISO-639. (The full list of the ISO-639 codes can be found at <http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt>.)

The second parameter is the country code which consists of two upper-case letters. The country code must conform to ISO-3166. (The full list of the ISO-3166 codes can be found at [http://www.chemie.fu-berlin.de/diverse/doc/ISO\\_3166.html](http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html).)

The **Locale** class also has a three parameter constructor. The third parameter is called the "variant code" and is usually used to distinguish the differences caused by a given computing platform. The variant codes conform to no standard, and are specific for the application where it is used and is a vendor or browser-specific code. The following example illustrates the creation of two locales with different variant codes "WIN" and "MAC". Utilizing a variant code can be useful if, for instance, font differences force different characters to be used on Windows versus Macintosh:

```
Locale winLocale = new Locale("en", "GB", "WIN");  
Locale macLocale = new Locale("en", "GB", "MAC");
```

The **Locale** class defines several static constants that give access to some commonly used locale instances. For example:

```
Locale locale = Locale.GERMAN;
```

is equivalent to the following:

```
Locale locale = new Locale("de", "DE");
```

and

```
Locale locale = Locale.GERMANY;
```

is equivalent to the following:

```
Locale locale = new Locale("de", "");
```

It should be taken into consideration while using locales, that a `Locale` object is just an identifier, which is passed to locale-sensitive objects such as **NumberFormat**, **DateFormat**, **Calendar**, **Collator**, etc., so, although it is possible to create a `Locale` object with any valid language and country code, the local-sensitive object may not know what to do with such a locale. As a result, locale-sensitive objects provide the **getAvailableLocales()** method, that returns an array of **Locale** objects that the object can process (generally speaking, the list of installed locales).

## Compound Messages

In addition to static text, it is sometimes needed to include a variable into a localized message. In this case parameterized (compound) messages are used. For example, we need to internationalize the following message:

At 12:30 PM on Jul 3, 2053, there was a disturbance in the Force on planet 7.

Date, time, planet number and event text ("a disturbance in the Force") are variables which can have different values during runtime execution. The **java.text.MessageFormat** class provides a solution that allows populating a localized string template with the appropriate values.

In the appropriate property file the following localized template for string given above (assume "str" is the key) is written:

```
str= At {1,time} on {1,date}, there was {2} on planet {0,number,integer}.
```

The expressions in braces specify the index of passed argument which will be substituted in the final string and optionally the format type and format styles.

First obtain a formatter instance for a given **Locale**.

```
Locale locale = Locale.US;  
MessageFormat formatter = new MessageFormat("");  
formatter.setLocale(locale);
```

Next obtain the template string from the properties file.

```
ResourceBundle resources = ResourceBundle.getBundle("MyResources", locale);  
String template = messages.getString("str");
```

Then define an array of arguments to be substituted into the localized string:

```
Object[] arguments = {  
    new Integer(7),  
    new Date(System.currentTimeMillis()),  
    "a disturbance in the Force"  
};
```

Finally, apply the pattern and substitute variables:

```
formatter.applyPattern(messages.getString("str"));  
System.out.println(formatter.format(msgArgs));
```

At the result localized string with the provided variable values is output.

## Overview of Internationalization in .NET

Internationalization on .NET platforms is based on the same common principles as it is in Java, however, the .NET i18n API and implementation are significantly different from the Java i18n API.

One of key elements of .NET internationalization support is the **System.Globalization.CultureInfo** class that holds culture-specific information, such as the associated language, sublanguage, country/region, calendar, and cultural conventions. This class also provides access to culture-specific instances of **DateTimeFormatInfo**, **DateTimeFormatInfo**, **NumberFormatInfo**, **CompareInfo**, and **TextInfo**. These objects contain the information required for culture-specific operations, such as casing, formatting dates and numbers, and comparing strings. The String class indirectly uses the **CultureInfo** class to obtain information about the default culture.

The **System.Resources.ResourceManager** class looks up culture-specific resources, provides resource fallback when a localized resource does not exist, and supports resource serialization. Using the methods of **ResourceManager**, a caller can access the resources for a particular culture using the **GetObject** and **GetString** methods. By default, these methods return the resource for the culture determined by the current cultural settings of the thread that made the call. (The thread class has a **CurrentUICulture** property which provides access to a **CultureInfo** object associated with a given thread.) A caller can use the **ResourceManager.GetResourceSet()** method to obtain a **ResourceSet**, which represents the resources for a particular culture, ignoring the culture fallback rules. The **ResourceSet** can then be used to access the resources, localized for that culture, by name.

## Simple Example

The following is a simple example illustrating localization on .NET:

```
using System;
using System.Globalization;
using System.Resources;

namespace MyNamespace
{
    public class HelloWorld
    {
        ResourceManager rm = ResourceManager.CreateFileBasedResourceManager
            ("MyStrings", "..\\..\\..\\Misc", null);

        // Get string according default CultureInfo.
        Console.WriteLine(rm.GetString("hello"));

        CultureInfo myClintl = new CultureInfo("de-DE", false);

        // Set "de-DE" CultureInfo for current thread.
        Thread.CurrentThread.CurrentUICulture = myClintl;

        // Print string value according to "de-DE" CultureInfo.
        Console.WriteLine(rm.GetString("hello"));
    }
}
```

## xTier™ i18n Service Overview

As demonstrated in the preceding section, Java and .NET use a conceptually similar approach to internationalization, but implement the supporting APIs in a significantly different manner. The xTier™ i18n Service provides a convenient and easy-to-use facility for application internationalization and localization, with a unified for Java and .NET. This means that the same approach, the same unified API and the same configuration property files can be used in both Java and/or .NET applications (or parts of applications). The xTier™ i18n Service's key features are:

- XML configuration, that provides a convenient facility and format for internationalization and localization.
- Unified API for Java and .NET. Internationalization mechanisms are absolutely the same for both platforms. This means that you don't need to do double work when internationalizing cross-platform application.
- All localized values are stored in one file. This is very convenient for persons who localize an application, because they do not need to constantly switch between different files.
- Direct API for compound messages formatting. This means that getting the value and formatting with specified parameters is performed with a single method call (remember that in Java this requires several steps, obtaining the value, then applying pattern and formatting).
- The xTier™ i18n Service provides the notion of "skins." Skins allow one property to have multiple values for the same property and locale. This provides an advanced level of customization for the user interface.
- Property groups allow logically organizing i18n properties.
- Variable substitutions allow a property value to refer to another property value. This allows the developer to utilize variable concepts with respect to internationalization.

- Automated configuration reloading. The application using the xTier™ i18n Service can automatically reload the i18n configuration from the XML file at a specified interval of time. In this way, i18n configuration changes can be made without restarting the application.

## Configuration

The xTier™ i18n Service is configured using the pre-defined `xtier_i18n.xml` configuration file. The formal specification for this file can be found in the `${XTIER_ROOT}/config/dtd/xtier_i18n.dtd` file. The xTier™ i18n Service region consists of the property group definition:

```
<group name="another.group">
  <!-- Locale-less property, -->
  <i18n name="file.path">
    <forall>somepath</forall>
  </i18n>
</group>
```

Groups allow i18n properties to be logically organized and additionally by using groups, property naming conflicts are prevented as properties with the same name be distinguished based upon their group membership. The `<group>` tag has a required attribute, **name**, which represents the unique name of the i18n group. This name can be used later at runtime to access this group's properties.

A group consists of the following i18n property definition:

```
<group name="example">
  <i18n name="hello">
    <!-- Skinless value for "en" language -->
    <locale lang="en">Hello World!</locale>
    <!-- Skinless value for "de" language -->
    <locale lang="de">Hallo Welt!</locale>
    <skin name="uppercase.skin">
      <!-- Skinned value for "en" language →
      <locale lang="en">HELLO WORLD!</locale>
      <!-- Skinned value for "en" language →
      <locale lang="en">HALLO WELTI!</locale>
    </skin>
  </i18n>
  <i18n name="prop">
    <locale lang="en">Now is {0, date}; {1}</locale>
    <locale lang="de">Jetzt {0, date}; {1}</locale>
  </i18n>
</group>
```

Each property is represented by an **<i18n>** tag, which has a required attribute **name**, which represents the name of the i18n property. Please note, this name should be unique within the i18n group. This name attribute can be used at runtime to access this property's value.

The **<i18n>** tag may include the **<forall>**, **<skin>** and **<locale>** tags. The **<forall>** tag defines the property value that is the same for any locale. This is a useful feature of the xTier™ i18n Service because it provides support for data that doesn't have a localized value. (This is opposed to Java with which it would be required to put the same value in each localized file for such properties.)

The **<locale>** tag defines the property value for a given locale. Language and country codes are specified by the **lang** (required) and **country** (optional) attributes. The **<locale>** value is used as a property value for the given locale. One of the key advantages of xTier™ i18n Service is that all localized values for a property are stored in one XML file. This makes the localization process much simpler, because the person who translates the property file does not need to constantly switch between different files (remember, that Java uses different files for each locale). When using xTier™ the translator performing the localization is always working with all values in a single file at the same time.

Skins provide the ability to have different values for the same property in a single locale. The values can be selected based upon specifying the appropriate skin to retrieve at runtime. In this way, there is an additional layer of user interface customization. For example, this is useful for displaying messages in short and in full-length forms dependent on settings or, as is demonstrated in the following configuration fragment, for displaying uppercase/lowercase messages (see the "hello" property definition). Additionally, as i18n properties can be used not only for textual messages, but also for storing different UI characteristics, such as color name, skins provide the ability to change the look and feel of the UI using the **I18nService.setSkin(String)**. The name of skin is specified by the **name** (required) attribute. This name can then be used at runtime to obtain the values for the corresponding skin.

The i18n configuration region also provides the ability to enable/disable and specify the time interval of the automated reloading. This facility is configured using the **<reload>** tag, which can be included within the **<region>** tag. The **<reload>** tag can have the values of **true** or **false**. If the value is **true**, the i18n configuration region automatically reloads with period, specified by the **msec** attribute. If the value is **false**, the i18n configuration region does not reload automatically. The reload interval is specified in milliseconds and if omitted, the default value of 600000 (10 min) is used. If the **<reload>** tag has a false value, this attribute is ignored. Note that in any case, the i18n configuration region can be reloaded using the **reload()** method.

```
<reload msec="1200000">true</reload>
```

## Properties

Each property in the **<locale>** and/or **<forall>** elements should contain a textual pattern that is a functional subset of the formatting pattern supported by the **MessageFormat** class. The following table shows the supported pattern elements as well as their .NET counterparts (i.e. the way the .NET service will interpret them):

Java {index[,formatType][,formatStyle]}	.NET {index[,alignment][:formatString]}
{N}	{N}
{N, number}	{N:G}
{N, number, integer}	{N:D}
{N, number, percent}	{N:P}
{N, number, currency}	{N:C}
{N, number, other}	{N:other}
{N, date}	{N:D}
{N, date, short}	{N:d}
{N, date, medium}	{N:d}
{N, date, long}	{N:D}
{N, date, full}	{N:G}
{N, date, other}	{N:other}
{N, time}	{N:T}
{N, time, short}	{N:t}
{N, time, medium}	{N:t}
{N, time, long}	{N:T}
{N, time, full}	{N:g}
{N, time, other}	{N:other}

Note that the .NET conversion of the formatting pattern does not guarantee identical output results. The i18n service makes the best effort to provide the closest formatting match between .NET and Java.

### Property Substitution

Any property in the `<locale>` and/or `<forall>` elements can include the `#{[group:]prop}` construct where **group** is an optional i18n group name and **prop** is the name of the property in that group. If the group is not specified the current group is assumed by default. The value of the referenced property will be substituted in place of `#{[group:]prop}`. The `#{[group:]prop}` construct can use forward declarations. This useful feature allows, for example, to repeatedly use one text fragment. Simply write the fragment as a property and then use a reference to this property everywhere it is needed.

## Usage

The way in which the developer uses the i18n service follows the standard pattern when using any xTier™ service: first obtain an instance of the xTier™ kernel that serves as a service registry. Using the instance of the xTier™ kernel an instance of any service can be obtained, in this case, the i18n service. Once the service instance is obtained the service's API can then be used. The i18n service provides two types of formatting methods: one that uses the current locale and another that takes a locale as a parameter. All methods accept from 0 to 5 parameters for convenience. If more than five parameters are required to format the message, the developer can use the method form that accept an object array. Note that the xTier™ i18n Service provides a direct API for formatting. This is contrasted to Java which requires several steps for such purposes; first a template is retrieved from resources, then set the formatter, and then format the message with the given parameters. The xTier™ i18n Service performs all of these operations automatically by simply calling the format method.

Using the current system locale:

- `format(String, String)`
- `format(String, String, Object)`
- `format(String, String, Object[])`
- `format(String, String, Object, Object)`
- `format(String, String, Object, Object, Object)`
- `format(String, String, Object, Object, Object, Object)`
- `format(String, String, Object, Object, Object, Object, Object)`

Using the supplied locale:

- `format(Locale, String, String)`
- `format(Locale, String, String, Object)`
- `format(Locale, String, String, Object[])`
- `format(Locale, String, String, Object, Object)`
- `format(Locale, String, String, Object, Object, Object)`
- `format(Locale, String, String, Object, Object, Object, Object)`
- `format(Locale, String, String, Object, Object, Object, Object, Object)`

The following example illustrates these basic steps:

```
// Get the instance of xTier kernel.  
XtierKernel xtier = XtierKernel.getInstance();
```

```
// Get the instance of 'i18n' service.  
I18nService i18n = xtier.i18n();
```

The next fragment retrieves and prints out the value of the property "hello" from the "example" group. (Refer to XML configuration example provided above to see how it is defined).

```
// Print value of property 'prop1' from 'example' group.
```

```
System.out.println(i18n.format("example", "hello"));
```

The following code example demonstrates skinned output. The skin should be set for the i18n service, and then if a given property has a value for the given skin, it will be retrieved. (Refer to configuration example above to see this property's XML definition.)

```
// Set skin 'uppercase.skin'.
i18n.setSkin("uppercase.skin");

// Print skinned value of property 'hello' from 'example'.
System.out.println(i18n.format("example", "hello"));
```

For the configuration provided above, this code will print "HELLO WORLD!" in uppercase. The following example demonstrates localized output:

```
// Print value of the same property for Locale.GERMANY locale.
System.out.println("prop2 with GERMANY locale: " + i18n.format(Locale.GERMANY, "example", "hello"));
```

This prints the value of the property "hello" for the **Locale.GERMANY** locale. As an alternative the locale can be set for the service:

```
// Set locale.
i18n.setLocale(java.util.Locale.GERMANY);
// Print value Locale.GERMANY locale.
i18n.format("example", "hello");
```

The following example demonstrates formatting:

```
// Print 'example:prop' value with parameters.
System.out.println("prop: " + i18n.format("example", "prop", new Date(), new String("Hello!")));
```

The parameters passed in will be used for message formatting. As demonstrated with this code snippet, formatting using the xTier™ i18n Service requires only a single method call, instead of the several separate steps used by Java.

Although the format method has different forms, that allow passing from 0 to 5 parameters, the developer is not limited with respect to the number of parameters as the i18n service provides a form that receives an object array. The same formatting as above is demonstrated using an object array:

```
// Print 'example:prop' property value using array of parameters.
Object[] params = new Object[] { new Date(), new String("Hello!")};
System.out.println("prop: " + i18n.format("example", "prop", params));
```

To format the property value for specified locale it is enough to specify the locale in method call.

```
// Print 'example:prop' with parameters for Locale.GERMANY locale.  
System.out.println("prop: " + i18n.format(Locale.GERMANY, "example", "prop", params));
```

Next code snippet demonstrates the usage of the configuration change listener and automated config reloading. In the listener any actions that should be performed upon configuration reloading can be specified.

```
// Reloading.  
i18n.addListener(new I18nChangeListener() {  
    public void i18nChanged() {  
        System.out.println("i18n configuration changed.");  
    }  
});  
  
i18n.reload();
```

## Conclusion

The xTier™ provides a convenient and easy-to-use i18n service for applications' internationalization and localization, unified for Java and .NET. Service implements some useful features, which make internationalization and localization processes much simpler and less labor-intensive, than if using built-in java or .NET internationalization mechanisms.

### Fitech Laboratories Inc.

#### Corporate Headquarters

300 Montgomery St., Suite 621  
San Francisco, CA 94104  
USA

Phone: 1-415-371-8234  
Fax: 1-415-371-8237

#### East Coast Sales Office

330 Madison Ave., 9th floor  
New York, NY 10017  
USA

Phone: 1-646-495-5076

#### Fitech Laboratories Japan

Toranomon40 MT Bldg. 3F  
5-13-1 Toranomon Minato-ku  
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711  
Web: [www.fitechlabs.co.jp](http://www.fitechlabs.co.jp)