CAPLIN

Caplin Trader 2.2

How To Add Localization Support

April 2011

CONFIDENTIAL

Contents

1 Preface

1.1 What this document contains

As supplied, **Caplin Trader** builds English language applications by default. This document explains how to code a Caplin Trader application so that it can subsequently be localized to other languages and local preferences (**locales**). It includes coding guidelines to help minimize problems and errors in the translation process.

About Caplin document formats

This document is supplied in three formats:

- Portable document format (*.PDF* file), which you can read on-line using a suitable PDF reader such as Adobe Reader®. This version of the document is formatted as a printable manual; you can print it from the PDF reader.
- Web pages (*.HTML* files), which you can read on-line using a web browser. To read the web version of the document navigate to the *HTMLDoc_m_n* folder and open the file *index.html*.
- Microsoft HTML Help (*.CHM* file), which is an HTML format contained in a single file.
  To read a *.CHM* file just open it – no web browser is needed.

For the best reading experience

On the machine where your browser or PDF reader runs, install the following Microsoft Windows® fonts: Arial, Courier New, Times New Roman, Tahoma. You must have a suitable Microsoft license to use these fonts.

Restrictions on viewing .CHM files

You can only read *.CHM* files from Microsoft Windows.

Microsoft Windows security restrictions may prevent you from viewing the content of *.CHM* files that are located on network drives. To fix this either copy the file to a local hard drive on your PC (for example the Desktop), or ask your System Administrator to grant access to the file across the network. For more information see the Microsoft knowledge base article at
http://support.microsoft.com/kb/896054/.

1.2 Who should read this document

This document is intended for Technical Managers and Software Developers who need to implement support for **localization** within a Caplin Trader application.

1.3 Related documents

- **Caplin Trader: Localization Overview And Concepts**

  Gives an overview of localization within Caplin Trader. It covers localization concepts, how support for localization is built in to Caplin Trader, and how Caplin Trader applications are localized.

- **Caplin Trader: How To Localize Your Application**

  Explains how to localize a Caplin Trader application to display text in a different language and display dates, times, number formats, and other items according to local preferences.

- **Caplin Trader: API Reference**

  The API reference documentation provided with Caplin Trader. The classes and interfaces of this API allow you to extend the capabilities of Caplin Trader.

1.4 Typographical conventions

The following typographical conventions are used to identify particular elements within the text.

Type	*Uses*
aMethod	Function or method name
aParameter	Parameter or variable name
/AFolder/Afile.txt	File names, folders and directories
`Some code;`	Program output and code examples
The `value=10` attribute is...	Code fragment in line with normal text
Some text in a dialog box	Dialog box output
`Something typed in`	User input – things you type at the computer keyboard
Glossary term	Items that appear in the “Glossary of terms and acronyms”
XYZ Product Overview	Document name
◆	Information bullet point
■	Action bullet point – an action you should perform

Note: Important Notes are enclosed within a box like this.
Please pay particular attention to these points to ensure proper configuration and operation of the solution.

Tip: Useful information is enclosed within a box like this.
Use these points to find out where to get more help on a topic.

Information about the applicability of a section is enclosed in a box like this.
For example: “This section only applies to version 1.3 of the product.”

1.5 Feedback

Customer feedback can only improve the quality of our product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Visit our feedback web page at https://support.caplin.com/documentfeedback/.

1.6 Acknowledgments

Adobe® Reader is a registered trademark of Adobe Systems Incorporated in the United States and/or other countries.

Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Java is a trademark or registered trademark of Sun Microsystems, Inc. in the U.S. or other countries.

2 Introduction

Applications written using the **Caplin Trader** framework, release 2.2 and later, are capable of displaying text to the end-user in a language other than English (the default language), with numbers and dates formatted for the displayed language. This is called localizing the application.

There are two stages to localizing a **Caplin Trader application**:

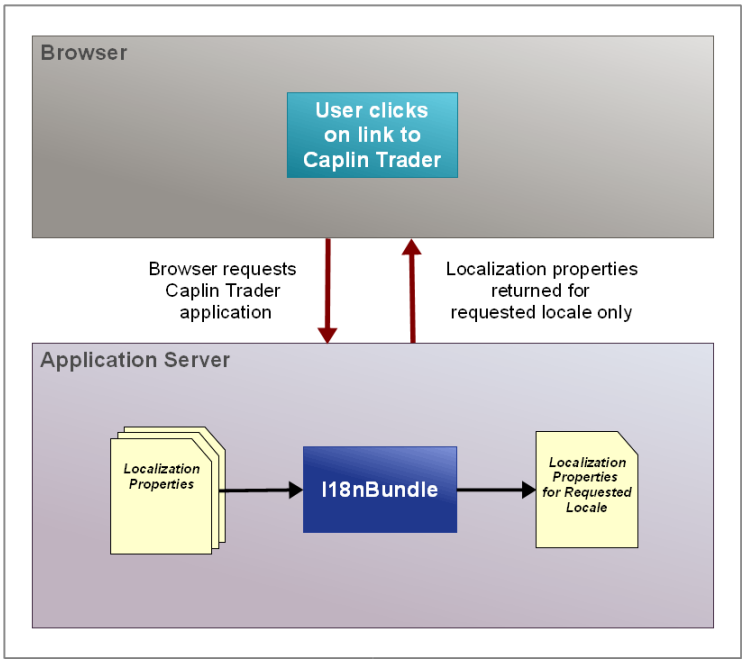
1. Write the application in a way that supports **localization**. This involves placing **localization keys** in the application code instead of the text that is displayed to the end-user. When the application runs, the Caplin Trader framework substitutes these localization keys with text in the required language.
2. Prepare **properties files** that contain language translations for each localization key. It is this translated (localized) text that the Caplin Trader framework displays to the end-user at run time. A set of properties files must be created for each language or **locale** that your application supports.

This document describes the first stage in this process: how to write an application that supports localization. The document **Caplin Trader: How To Localize Your Application** describes the second stage: how to localize an application for a particular locale.

For an overview of localization concepts, see the document **Caplin Trader: Localization Overview And Concepts**.

2.1 How Localization Works

I18nBundle is a Java servlet that resides on the application server, and runs when the server receives a browser request for the Caplin Trader application. Its function is to return the **localization properties** that map **localization keys** to the localized text that is displayed in the browser.



I18N bundles localization properties for the requested locale

3 Assumptions and restrictions

The Caplin Trader framework

The installed Caplin Trader framework must be at release 2.2 or later, and Caplin Trader applications that support localization must be built using this framework.

File paths referred to in this document

For convenience, this document assumes that the Caplin Trader framework, Caplin Trader **blades** code, and your application code are held in a software repository, and installed to the following directory on a Windows PC:

`X:\scm_root\NovoBank` (where `X` is the drive mapping)

It also assumes that your application is based on the Reference Implementation of Caplin Trader, which is supplied with the Caplin Trader installation kit.

Locale identifiers

In Caplin Trader each locale is defined by a **locale identifier**, such as:

`en_US`	American English
`en_GB`	British variant of English
`fr_FR`	Standard French
`fr_CA`	Canadian variant of French
`cz_CZ`	Czech

Locale identifiers are of the form:

`<language-code>_<country>`

where:

- `<language-code>` follows the ISO 639.1 standard for 2 letter language codes.
- `<country>` follows the ISO 3166 standard for 2 letter country codes.

4 Adding localization support to an application

Before a Caplin Trader application can be localized for a particular locale, the application code must be written in a way that supports localization. This applies to both existing and new applications.

4.1 Localization properties files

Localization properties files group together the **localization properties** for a particular locale. Each localization property sets the value of a **localization key**, and there is a set of **properties files** for each supported **locale**. Localization properties define the following:

- The localized text that is displayed on the screen to the end-user.
- Date formats.
- Number formats.

The Caplin Trader framework ships with properties files for the English language:

Filename	Content
en.properties	Localization properties for the American variant of the English language.
en_GB.properties	Localization properties for the British variant of the English language.

Your installed version of the Caplin Trader framework may have properties files for other locales (see the release notes for further details).

Character Encoding

Localization properties files must be **UTF-8** encoded, and must not have any **Byte-Order Mark (BOM)** start bytes.

Location of properties files

The Caplin Trader framework

Localization properties files are shipped with the following software modules of the Caplin Trader framework.

- Framework libraries
- **Blades**
- Caplinx (the code for the Reference Implementation of Caplin Trader)

By convention, properties files are located in subdirectories of the software module, and the path to each file takes the general form:

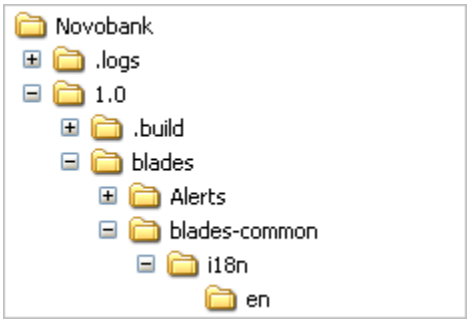
.../i18n/en/en.properties.

For example, if Caplin Trader blades code is checked in to a repository at:

Novobank/1.0/blades

then the American English properties file for blades-common software is located at:

Novobank/1.0/blades/blades-common/i18n/en/en.properties



The properties files supplied with the Caplin Trader framework must not be modified or deleted. When you localize your application, you must provide localizations for the properties defined in these files (see **Caplin Trader: How To Localize Your Application** for further information).

Application properties files

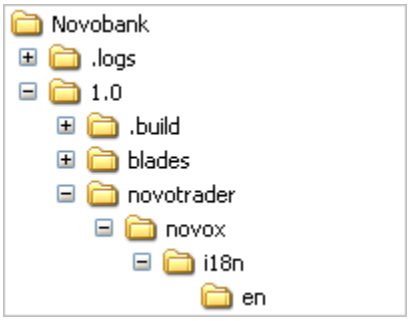
When you create a properties file for a new or existing Caplin Trader application, it must be stored in a file path that does not get overridden when the Caplin Trader framework is upgraded.

For example, if your application code is checked in to a repository at:

Novobank/1.0/novotrader/novox

then the recommended location of English properties files is:

Novobank/1.0/novotrader/novox/i18n/en



This ensures that your properties files do not get overridden when you build your application or upgrade the Caplin Trader framework.

Content of properties files

Localization properties files can contain any of the following:

- Lines that define localization properties (one property per line)
- Comment lines
- Blank lines

A comment line must start with the `#` character and can contain any text.

Example comment line

```
# This is a comment.
# A comment can contain white space, and punctuation such as - , and !
```

Lines that define localization properties use the following notation:

Syntax of a localization property

```
<localization_key> = <localization_value>
```

In the notation shown above, `<localization_key>` is the name of the key and `<localization_value>` the value assigned to the key. Note that the space before and after the `=` sign is optional.

The name of a localization key can be assigned to a namespace.

Example of a namespaced localization key

```
novox.layout.name.foreign_exchange=Foreign Exchange
```

In this case, the localization key is in the `novox.layout.name` namespace. When you create **keys** for your own application, you should place each key in a unique namespace (such as `novox`). This ensures that localization keys in your application do not have the same name as localization keys supplied with the Caplin Trader framework.

The value of a localization key can be any string of characters, including white space and punctuation. In the example above, the value of the key is set to `Foreign Exchange`.

Using variables in a key value

The value of a key can specify one or more variables, by enclosing the name of each variable inside [ ] characters.

Example localization value that takes variables

```
novox.account.message=Your account has [amount] dollars available.
```

In the example above, the value of the key specifies the named variable `amount`.

When a localization key takes named variables, the application code that uses the key must provide a value for each named variable (see Defining localizable items in JavaScript 10).

Note: Variables cannot be used when the localization key is referred to in XML or HTML templates.

Adding a leading or trailing space to a key value

To add a leading or trailing white space to a key value, escape the space with a \ character. The following example adds a leading space to a key value.

Localization key value with leading space

```
novox.example.message=\ This text has a leading space.
```

Splitting key values over more than one line in the properties file

To split a key value over more than one line, add a space followed by the \ character at the end of the line where the line splits.

Splitting a key value over two lines

```
novox.example.message=This text value \
is split over two lines.
```

Line splits only affect the text in the properties file, and not how the text is displayed on the screen.

Date and number properties

Properties files are shipped with the Caplin Trader framework that define localization keys and English settings (American and British) for the following date and number properties:

- Long and short month names (for example `ct.i18n.date.month.january=January`)
- Long and short day names (for example `ct.i18n.date.month.monday=Monday`)
- Long and short date formats (for example `ct.i18n.date.format=d-m-Y`)
- Time format separator (as in 04:30)
- Decimal radix character (as in 4.2)
- Number grouping separator (as in 3,200,000)
- Number multipliers (as when 5k represents 5,000)

Localization keys for date and number properties can be used in your application code. If you localize your application for a locale that is not already supported by the Caplin Trader framework, the English values of these keys must be translated for that new locale.

4.2 Defining localizable items in JavaScript

The **caplin.i18n.Translator** class of the Caplin Trader API allows you to define items in JavaScript that support localization. The way you define these items depends on the item you want to localize.

Generally speaking, exceptions, global system errors, and logs do not require **localization support**. On the other hand, if an exception is displayed to the end-user then the exception text must be coded in a way that supports localization.

Take for example an input validation system that throws an exception if a user enters a word into a number field. If that exception is caught by a display manager that displays the exception to the end-user, the exception text must be coded in a way that supports localization.

Classes in the **caplin.element.formatter** and **caplin.element.parser** namespaces also support localization. These classes are normally part of element renderers and configured in XML, but you can call these classes from the JavaScript code of your application.

Displaying strings

The global JavaScript function `ct.i18n()` returns the localized value of a localization key. This allows you to place localization keys in your JavaScript code instead of the text that is displayed on the screen.

The following example shows how JavaScript code that displays text in an alert box can be modified to support localization. Although the code in this example is simple, the same process can be applied to code that uses the classes and methods of the Caplin Trader API to display text to the end-user.

Here is the example JavaScript code that must be modified to support localization.

Alert code to be modified

```
alert("An error occurred, please try again later.");
```

To add localization support to this code, first create a property that sets the value of a localization key to the alert text you want to display, and save this to the American English version of your application properties file (*en.properties*).

Create a localization property in *en.properties*

```
novox.error.message=An error occurred, please try again later.
```

You can now use this localization key in the JavaScript code that displays the alert message.

Alert code that supports localization

```
alert( ct.i18n("novox.error.message") );
```

The JavaScript code for the alert message now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the code shown above will display the alert message in Spanish.

For further information about the `ct.i18n()` function, see the description of the **caplin.i18n.Translator** class in the **Caplin Trader API Reference** document.

Displaying strings with named variables

The global JavaScript function `ct.i18n()` returns the localized value of a localization key when the string value of the key specifies named variables. In this case the `ct.i18n()` function must be passed two arguments:

- The name of the localization key.
- A JavaScript object containing the name and value of each named variable.

The following example shows how JavaScript code that displays text in an alert box can be modified to support localization.

Alert code to be modified

```
var amount=20000;
alert("Your account has " + amount + " dollars available.");
```

To add localization support to this code, first create a property that sets the value of a localization key to the alert text you want to display, and save this to the American English version of your application properties file (*en.properties*).

Create a localization property

```
novox.funds.message=Your account has [amount] dollars available.
```

When the value of a localization key specifies a named variable, the name of the variable must be placed inside [ ] characters. In this case the name of the variable is `amount`.

You can now use this localization key in the JavaScript code that displays the alert message.

Alert code that supports localization

```
alert( ct.i18n("novox.funds.message", {amount: 20000}) );
```

Because the value of the localization key specifies a named variable, the name and value of this variable must be passed to the `ct.i18n()` function as a JavaScript object. In this case the name of the variable is `amount` and its value is 20000, so the object passed in is `{amount: 20000}`.

The JavaScript code for the alert message now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the code shown above will display the alert message in Spanish.

If the name and value of the `amount` variable are not passed to the `ct.i18n()` function, the `[amount]` characters are treated as plain text and not as a reference to a variable.

Omitting the name and value of the variable

```
alert( ct.i18n("novox.funds.message") );
```

In this case the text of the alert message is the literal string
`"Your account has [amount] dollars available."`.

For further information about the `ct.i18n()` function, see the description of the **caplin.i18n.Translator** class in the **Caplin Trader API Reference** document.

Displaying dates

The `formatDate()` method of the **caplin.i18n.Translator** class formats a date to the pattern specified by a date format string.

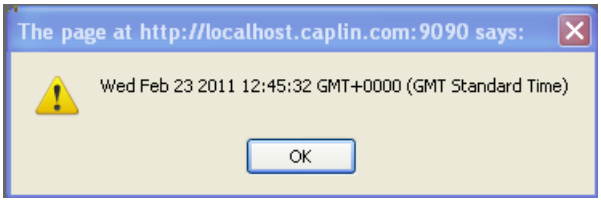
If a date format string is not passed in as an argument to this method, the date is formatted to a pattern specified by the localization key `ct.i18n.date.format`.

The following example shows how JavaScript code that displays a date in an alert box can be modified to support localization using this method.

Alert code to be modified

```
alert( new Date() );
```

The alert displayed on the screen would look like the following:



To add localization support for this alert, place the code `new Date()` inside a call to the `formatDate()` method.

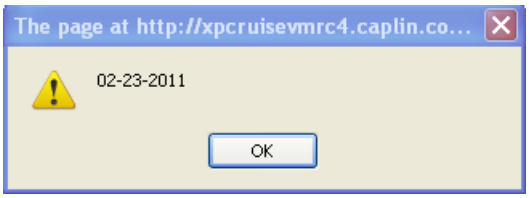
Because a localization key that defines the date format string is supplied with the Caplin Trader framework (`ct.i18n.date.format`), there is no need to pass a date format string to the `formatDate()` method.

Alert code that supports localization

```
alert( caplin.i18n.getTranslator().formatDate(new Date()) );
```

The JavaScript code for the alert message now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the code shown above will display the alert message using the Spanish value of the date format string.

Because the American English property for the `ct.i18n.date.format` key sets the date format string to `m-d-Y`, the alert message would look like the following if `en_US` was the current locale:



For further information about the `formatDate()` method of the **caplin.i18n.Translator** class, see the **Caplin Trader API Reference** document.

Displaying part of a date

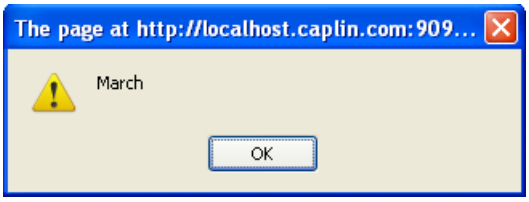
The `formatDate()` method of the **caplin.i18n.Translator** class formats a date to the pattern specified by a date format string argument. For example, when `F` is passed in as the date format string, the method returns the month part of the date (as defined by the localization property for that month).

The following example displays the localized name of the current month in an alert box.

Alert code that supports localization

```
alert( caplin.i18n.getTranslator().formatDate(new Date(), "F") );
```

For example, if English is the current locale and the current month is March, the `formatDate()` method returns the value defined by the localization key `ct.i18n.date.month.march`. In this case the alert message would look like the following:



The same approach can be applied to short month abbreviations, and long and short day names, by passing in the appropriate date format string argument.

Format string	Localization key that determines the localized value	Description
`F`	`ct.i18n.date.month.<month-name>`	Long month name (such as March)
`M`	`ct.i18n.date.month.short.<month-name>`	Short month name (such as Mar)
`l`	`ct.i18n.date.day.<day-name>`	Long day name (such as Monday)
`D`	`ct.i18n.date.day.short.<day-name>`	Short day name (such as Mon)

For further information about the `formatDate()` method of the **caplin.i18n.Translator** class, and the permitted format string arguments, see the **Caplin Trader API Reference** document.

Displaying time

The `formatTime()` method of the **caplin.i18n.Translator** class uses a time format separator to format a number as time. Because the time format separator is defined by a localization key, the `formatTime()` method already supports localization.

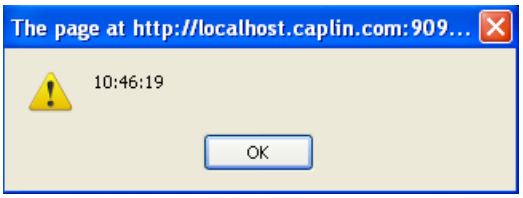
Localization key and English value	Description
`ct.i18n.time.format.separator=:`	The time format separator (as in 10:22)

The following example shows a number being passed to the `formatTime()` method. In this case the returned value is displayed in an alert box.

Formatting a number as an amount

```
alert( caplin.i18n.getTranslator().formatTime(104619) );
```

Because the current locale is American English, the `formatTime()` method uses `:` as the time format separator. The alert displayed on the screen would look like the following:



The `formatTime()` method also formats numbers containing four numerical characters (such as `1046`, which would be formatted as `10:46`). The method throws an exception if the number passed in contains more than six digits, or contains characters that are not numerical.

For further information about the `formatTime()` method of the **caplin.i18n.Translator** class, see the **Caplin Trader API Reference** document.

Displaying the current time

The `formatDate()` method of the **caplin.i18n.Translator** class formats a date to the pattern specified by a date format string argument, and can be used to display the current time.

The following example shows JavaScript code that uses this method to display the current time in an alert box.

Displaying the current time in an alert box

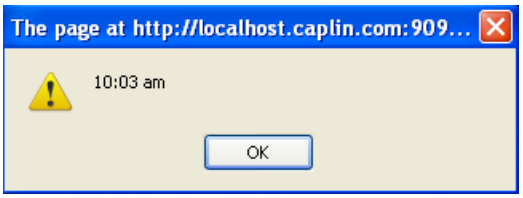
```
alert( caplin.i18n.getTranslator().formatDate(new Date(), "g:i a") );
```

In this example, the `formatDate()` method is passed a format string that is constructed from the following format characters:

Format string characters

Character	Description
`g`	The 12-hour format of the hour, without leading zeros.
`i`	The minutes past the hour, with leading zeros.
`:`	The character that separates hours from minutes.
`a`	The lowercase 12-hour clock notation (am or pm).

The alert displayed on the screen would look like the following:



To add localization support to this code, replace the `:` format character in the date format string with the localized time format separator. The localized time format separator is returned by the global `ct.i18n()` function when it is passed the localization key `ct.i18n.time.format.separator`.

Alert code that supports localization

```
var localizedTimeFormatSeparator = ct.i18n("ct.i18n.time.format.separator");
var timeFormatString = "g" + localizedTimeFormatSeparator + "i a";
alert( caplin.i18n.getTranslator().formatDate(new Date(), timeFormatString) );
```

The JavaScript code for the alert message now supports localization.

For further information about the `formatDate()` method of the **caplin.i18n.Translator** class, the permitted format string arguments, and the global `ct.i18n()` function, see the **Caplin Trader API Reference** document.

Displaying numbers

The `formatNumber()` method of the **caplin.i18n.Translator** class formats a number using the following localization keys:

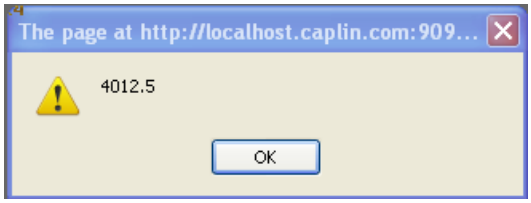
Localization key and English value	Description
`ct.i18n.decimal.radix.character=.`	Decimal radix character (as in 4.2)
`ct.i18n.number.grouping.separator=,`	Number grouping separator (as in 3,200,000)

The following example shows how JavaScript code that displays a number in an alert box can be modified to support localization using this method.

Alert code to be modified

```
alert( 4012.5 );
```

The alert displayed on the screen would look like the following:



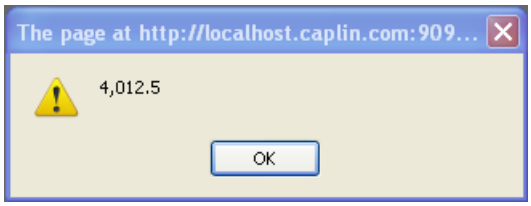
To add localization support for this alert, place the number `4012.5` inside a call to the `formatNumber()` method.

Alert code that supports localization

```
alert( caplin.i18n.getTranslator().formatNumber(4012.5) );
```

The JavaScript code for the alert message now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the code shown above will display the alert message using the Spanish radix character and grouping separator.

The alert message would look like the following if English was the current locale:



For further information about the `formatNumber()` method of the **caplin.i18n.Translator** class, see the **Caplin Trader API Reference** document.

Setting the number grouping separator to a space

Some locales use white space as the number grouping separator (as in `22 359`). To set a space character as the number grouping separator, escape the space with the `\` character in the property that defines the number grouping separator.

Setting a key value to white space

```
ct.i18n.number.grouping.separator=\<white space character>
```

In the example above, `<white space character>` represents the space character.

For further information, see the document **Caplin Trader: How To Localize Your Application**.

Customized number formatting

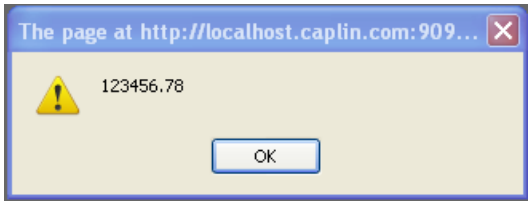
Sometimes it is necessary to customize the way that a number is formatted.

The following example shows how JavaScript code that displays a floating point number, but that has no number grouping separators, can be modified to support localization.

Alert code to be modified

```
alert( 123456.78 );
```

The alert displayed on the screen would look like the following:

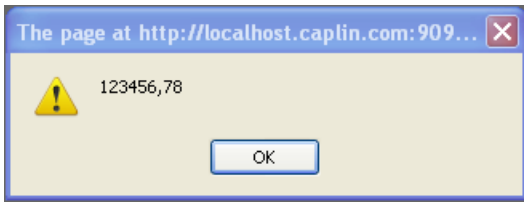


To add localization support for this alert, replace the decimal point with the localized decimal radix character.

Alert code that supports localization

```
var number = 1234567.98
var localNumber =
    (number + "").replace(".", ct.i18n("ct.i18n.decimal.radix.character"));
alert( localNumber );
```

The JavaScript code for the alert message now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the alert message would look like the following:



This example also assumes `ct.i18n.decimal.radix.character=,` in the Spanish properties file (*es.properties*).

For further information about the `ct.i18n()` function, see the description of the **caplin.i18n.Translator** class in the **Caplin Trader API Reference** document.

Displaying numbers using a formatter class

Classes of the Caplin Trader API that format numbers are called by the Caplin Trader framework when they are part of an element renderer that renders data in display component. Element renderers are configured in XML, but you can also call formatter classes from the JavaScript code in your application.

Displaying amounts

The `format()` method of the **caplin.element.formatter.AmountFormatter** class formats a number as an amount with a trailing amount suffix that represents either a thousands, millions, or billions multiplier. Because amount suffixes are defined by localization properties, the `format()` method already supports localization.

Amount suffixes

Localization key and English value	Description
`ct.element.number.formatting.amount.suffix.short.thousands=K`	The amount suffix that represents one thousand (for example 5K = 5,000).
`ct.element.number.formatting.amount.suffix.short.millions=M`	The amount suffix that represents one million (for example 5M = 5,000,000).
`ct.element.number.formatting.amount.suffix.short.billions=B`	The amount suffix that represents one billion (for example 5B = 5,000,000,000).

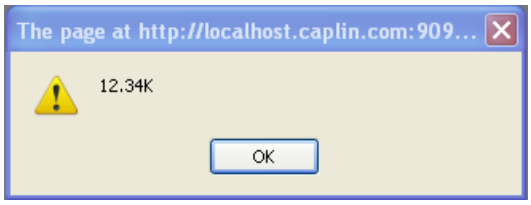
The following example shows a number being passed to the `format()` method when English is the current locale. The formatted number in this example is displayed in an alert box.

Formatting a number as an amount

```
alert( caplin.element.formatter.AmountFormatter.format(12340, {}) );
```

Because English is the current locale, the `format()` method returns a floating point number and the K amount token suffix (the amount token representing the one thousand multiplier).

The alert displayed on the screen would look like the following:



If the number passed to the `format()` method is less than one thousand, the number is returned without an amount token suffix.

For further information about the **caplin.element.formatter.AmountFormatter** class, see the **Caplin Trader API Reference** document.

Parsing numbers

Classes of the Caplin Trader API that parse numbers are called by the Caplin Trader framework when an end-user enters data into the input control of a display component. Input controls use element renderers that are configured in XML, but you can also call parser classes from the JavaScript code in your application.

Parsing floating point numbers

The `parse()` method of the **caplin.element.parser.ThousandsParser** removes the number grouping separator from a floating point number. Because the number grouping separator and decimal radix character are defined by localization properties, the `parse()` method already supports localization.

Localization key	Description
`ct.i18n.decimal.radix.character`	Decimal radix character (as in 4.2)
`ct.i18n.number.grouping.separator`	Number grouping separator (as in 3,200,000)

The following example shows a floating point number being passed to the `parse()` method when English is the current locale.

Parsing a floating point number

```
caplin.element.parser.ThousandsParser.parse( "5,000,000.0", {} );
```

Because English is the current locale, the `parse()` method removes the `','` character and returns the number `5000000.0`.

If the number passed to the parse method contains the wrong number grouping separator or the wrong decimal radix character for the current locale, the number is returned unchanged.

For further information about the **caplin.element.parser.ThousandsParser** class, see the **Caplin Trader API Reference** document.

Parsing amount suffixes - the AmountParser class

The `parse()` method of the **caplin.element.parser.AmountParser** returns a number if it is passed in a number and an amount suffix. Because amount suffixes are defined by localization properties, the `parse()` method already supports localization.

Amount token suffixes

Localization key and English value	Description
`ct.element.number.formatting.amount.suffix.short.thousands=K`	The amount suffix that represents one thousand (for example 5K = 5,000).
`ct.element.number.formatting.amount.suffix.short.millions=M`	The amount suffix that represents one million (for example 5M = 5,000,000).
`ct.element.number.formatting.amount.suffix.short.billions=B`	The amount suffix that represents one billion (for example 5B = 5,000,000,000).

Amount suffixes are not case sensitive.

The following example shows a number with an amount token suffix being passed to the `parse()` method when English is the current locale.

Parsing a number that has an amount token suffix

```
caplin.element.parser.AmountParser.parse( "5K", {} );
```

Because English is the current locale, the `parse()` method multiplies `5` by one thousand (the multiplier for the amount token `K`), and returns the number `5000`.

If the amount passed to the `parse()` method contains an invalid amount token, or if the prefix to the amount token is not a number, null is returned. When you localize your application you can set different values for these localization keys, but you cannot add new keys or change the multiplier value of these keys.

For further information about the **caplin.element.parser.AmountParser** class, see the **Caplin Trader API Reference** document.

Parsing amount suffixes - the LocalisedAmountParser class

The `parse()` method of the **caplin.element.parser.LocalisedAmountParser** returns a number if it is passed in a number and an amount suffix. Because amount suffixes are defined by localization properties, the parse method already supports localization.

Amount token multiplier suffixes

Localization key and English value	Description
`ct.element.number.formatting.multiplier.k=1000`	The multiplier when k or K is the amount suffix (for example 5k = 5 x 1000).
`ct.element.number.formatting.multiplier.m=1000000`	The multiplier when m or M is the amount suffix (for example 5m = 5 x 1000000).
`ct.element.number.formatting.multiplier.b=1000000000`	The multiplier when b or B is the amount suffix (for example 5b = 5 x 1000000000).

The following example shows a number and amount suffix being passed to the `parse()` method when English is the current locale.

Parsing a number that has an amount token multiplier

```
caplin.element.parser.LocalisedAmountParser.parse( "5M" );
```

Because English is the current locale, the `parse()` method multiplies `5` by one million (the multiplier when the amount suffix is `m` or `M`), and returns the number `5000000`.

If the amount passed to the `parse()` method contains an invalid amount suffix, or if the prefix to the amount suffix is not a number, null is returned.

When you localize your application you can set different multiplier values for these keys, and add new keys that have other multiplier values. This is useful for languages that use different amount suffixes and multipliers.

For further information about the **caplin.element.parser.LocalisedAmountParser** class, see the **Caplin Trader API Reference** document.

4.3 Defining localizable items in Configuration XML

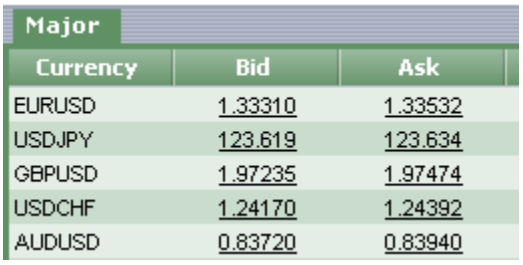
The display components of a Caplin Trader application display information to the end-user, and allow the end-user to interact with the application (for example, by clicking a button or entering a number). The information that is displayed on the screen may be in the form of text, dates, or numbers.

Most display components (such as grids and tiles) are configured in XML, as is the layout of these components on the screen. Before you can localize the application, you must make sure that the configuration XML supports localization.

XML that defines static text

Configuration XML can contain static text that is displayed on the screen to the end-user. To add localization support to this XML, you must replace this static text with **localization tokens**. Each localization token identifies a localization key, and when the Caplin Trader framework parses the configuration XML, it replaces each localization token that it finds with the localized value of the localization key, which is the text that is displayed on the screen.

The following example shows how XML that configures a grid display component can be modified to support localization. Here is what the grid looks like on the screen:



Major

Currency	Bid	Ask
EURUSD	1.33310	1.33532
USDJPY	123.619	123.634
GBPUSD	1.97235	1.97474
USDCHF	1.24170	1.24392
AUDUSD	0.83720	0.83940

A Grid displaying prices for currency pairs

The rows of the grid contain prices for currency pairs. While this part of the grid is configured in XML, it does not need to be modified, as the information you see on the screen is generated dynamically at run time.

The static text displayed in the column headers (Currency, Bid, and Ask), and the text that identifies the name of the grid (Major), is also defined in XML. It is this XML that must be modified, by replacing the static text with localization tokens.

Here is what the XML that must be modified looks like for the Currency column:

XML to be modified (Currency column)

```
<column id="description"
fields="InstrumentDescription"
displayName="Currency"
width="80"
mandatory="true"
primaryFieldType="text"/>
```

The `displayName` attribute of the `<column>` tag sets the header text of the column. In this case the header text is set to 'Currency'.

To add localization support to this code, first create a property that sets the value of a localization key to the text that you want as the column header, and save this to the American English version of your application properties file (*en.properties*).

Create a localization property

```
novox.column.header.currency=Currency
```

To create a localization token from this localization key, place the name of the localization key inside curly braces `{}` that are prefixed by the `@` character. This localization token can now be used in the XML that defines the header text for the Currency column.

XML that supports localization (Currency column)

```
<column id="description"
fields="InstrumentDescription"
displayName="@{novox.column.header.currency}"
width="80"
mandatory="true"
primaryFieldType="text"/>
```

The XML configuration for the Currency column header text now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the header text of the Currency column will be displayed in Spanish.

In a similar manner, localization tokens also need to be created and inserted in the configuration XML for the static text of the Bid and Ask columns headers, and for the text that identifies the name of the grid (Major).

When creating a new application, or when modifying an existing application, you must make sure that the configuration XML supports localization, by replacing the static text that is displayed on the screen with localization tokens, as shown in the example above.

Tip: Take care not to create localization tokens for XML attributes that do not define displayable text. For example, `primaryFieldType="text"` in the example above does ***not*** need to be support localization.

4.4 Defining localizable items in the application database

When a Caplin Trader application is first built, XML from source layout configuration files is loaded into an application database. When the Caplin Trader application runs in a browser, it is the XML held in the database that is served by the application server, and not the XML in the source layout configuration files. XML is served in this way so that end-users can save the custom layouts they create.

If you are converting an existing application to support localization, there are two ways to modify the XML held in the application database:

1. Re-load the application database with XML that has already been modified to support localization (see Defining localizable items in Configuration XML 23).
2. Modify the XML held in the database to support localization, by replacing the static text that is displayed on the screen with localization tokens.

Advantages and Disadvantages of each option

Option 1: The advantage of re-loading the database is that it is easy to implement (you simply run an ant task). The disadvantage is that the custom layouts saved by end-users will be lost.

Option 2: The advantage of modifying the XML held in the database is that the custom layouts saved by end-users will be preserved. The disadvantage is that you will have to find a way of doing this for the particular database you are using (there is no ant task that you can run).

The following instructions assume that your application code and build files are installed to the following directory on a Windows PC:

X:\scm_root\NovoBank\1.0\novotrader (where *X:* is the drive mapping)

This directory also contains the top level *build.xml* file for your application.

Re-loading the application database

To reload the application database:

- Open a command prompt window.
- Navigate to the directory that contains the top level *build.xml* file for your application. (for example, *X:\scm_root\NovoBank\1.0\novotrader*)
- Run the command `ant webcentric-reset`

The application database will now be re-loaded with XML from the source layout configuration files.

Note: Re-loading the application database destroys custom layouts that end-users have saved.

Identifying the XML that is held in the application database

If your application is based on the Reference Implementation of Caplin Trader, XML from the following layout configuration files is loaded into the application database when the application is first built:

Source of the XML held in the application database

Source file	Description
caplintrader.xml	Contains the text headers, labels and captions that are used by layout elements, and text for the dialogs that are available in the application.
<XXX>_Layout.xml (where *<XXX>* is a unique identifier for a layout)	Contains caption text for the panels used in a particular layout (such as the panels used by the default FX and FI layouts). Panels contain display components, such as grids and charts.

The files in the table above can be found in sub directories of *NovoBank\1.0\novotrader\build\xml*.

4.5 Defining localizable items in HTML

HTML templates can be used to construct display components, or to modify the appearance of display components according to the theme selected by the end-user. HTML templates are used in:

- The Reference Implementation of Caplin Trader
- Caplin supplied blades

The HTML templates provided with the Caplin Trader installation kit already support localization. If your application uses custom HTML templates, you will need to add localization support to these templates for any text that is displayed on the screen.

The following example shows how HTML can be modified to support localization. Here is what the HTML that must be modified looks like:

HTML to be modified

```
<dt>Near Date:</dt>
```

In this case the text `Near Date:` is enclosed by `<dt></dt>` tags, which means that it is an item in a list.

To add localization support to this HTML, you must replace this text with a localization token. The localization token identifies a localization key, and when the Caplin Trader framework parses the HTML, it replaces each localization token that it finds with the localized value of the localization key (the text that is displayed on the screen).

First create a property that sets the value of a localization key to the text you want to display, and save this to the American English version of your application properties file (*en.properties*).

Create a localization property

```
novox.theme.fx.ticket.near_date=Near Date:
```

To create a localization token from this localization key, place the name of the localization key inside curly braces `{}` that are prefixed by the `@` character. This localization token can now be placed inside the `<dt></dt>` tags in the HTML.

HTML that supports localization

```
<dt>@{novox.theme.fx.ticket.near_date}</dt>
```

The HTML now supports localization. For example, if the application is later localized for Spanish and `es_ES` is the current locale, the text will be displayed in Spanish.

5 Ensuring a good quality translation

When you add localization support to your application, you need to be careful about the way variables are used in localization property definitions (see Displaying strings with named variables 11). The following examples show some of the problems that can arise when variables are used in this way.

Consider the string:

"The duration is [interval] hours."

When the variable "interval" is 1, the substitution is grammatically incorrect:

"The duration is 1 hours."

It is not easy to correct the grammar in this case, and situations like this should be avoided if at all possible. In other situations the English text may be grammatically correct, but when the application is localized for another locale, the translated text may not make sense.

Consider the string:

"Open the [object]"

In English the definite article 'the' is the same no matter what the object is, but in many languages (such as German) this is not the case:

English	German translation	Comment
Open the door	Öffnen Sie **die** Tür	The definite article "die" is correct.
Open the suitcase	Öffnen Sie **den** Koffer	The definite article "die" would be grammatically incorrect.

This problem is easily fixed (once you are aware of it), by changing the scope of the substitution:

"Open [theobject]"

In this case the variable now includes the definite article, but more subtle problems may arise that are not found until translation time, when localizing the application for a different locale.

Recommendation for variable substitution

Do not use variables in text strings unless the variable is a proper noun (names of people, places, and so on). If this rule is adopted, problems with variable substitution will be avoided.

6 Tools for implementing localization support

Caplin supplies a number of reporting tools that identify the localization keys used in your application. While these reporting tools are primarily designed to help you localize your application for a new locale, you will also find them useful when adding localization support to your application.

For further information about these reporting tools, see the document **Caplin Trader: How To Localize Your Application**.

7 Glossary of terms and acronyms

This section contains a glossary of terms, abbreviations, and acronyms relating to localization support in Caplin trader applications.

Term	Definition
Blade	A business component that provides domain specific functionality in a **Caplin Trader application**. Each Caplin Trader blade implements a small, well-defined set of closely related functions.
Byte-Order Mark (BOM)	The Byte-Order Mark (BOM) is a Unicode character used to signal the byte order of a text file or stream. Byte order has no meaning in **UTF-8**, so a Byte-Order Mark only serves to identify a file or text stream as UTF-8 encoded, or to identify that it was converted from another format that has a Byte-Order Mark. In Caplin Trader, **localization properties files**, though encoded in UTF-8, must not contain Byte-Order Marks. [Definition adapted from Wikipedia contributors, "Byte order mark," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/wiki/Byte_order_mark (accessed April 6, 2011)]
Caplin Trader	A web application framework for constructing browser-based financial trading applications.
Caplin Trader application	A browser-based client application that has been built using **Caplin Trader**.
i18n	Abbreviation for **internationalization** (“i<18chars>n”).
Internationalization	An alternative term for the process of adding **localization support** to a software application.
L10n	Abbreviation for localization (“L<10chars>n”).
Locale	The aggregate of a software user's language, country, and any special variant preferences that the user wants to see in their user interface (such as particular date formats and number formats). Such an aggregate is uniquely identified by a **locale identifier**.
Locale identifier	The unique identification of a **locale**. In Caplin Trader, locale identifiers are of the form: `<language-code>_<country>` For example: `en_US` (American English) or `en_GB` (British English).
Key	In this document, this term is short for **localization key**.
Localization	The process of implementing a particular **locale** within a software application. This typically involves: ◆ Translating text that appears on screen, and in reports and logs, into the language of the locale. ◆ Defining the formats of dates, numbers, currencies, and so on, according to the requirements of the locale.
Localization key	The string of characters that identifies a particular **localization property**. For example: `blade.fxtile.buy` A localization key can belong to a **localization namespace**. For more information, see the document **Caplin Trader: Localization Overview And Concepts**.

Term	Definition
Localization namespace	A namespace convention and standard used for **localization keys** that partitions the keys according the particular areas of a Caplin Trader application they apply to. For more information, see the document **Caplin Trader: Localization Overview And Concepts**.
Localization property	A **localization key** and the value of that key in a given locale. A localization property has the general form: `<keyname>=<value>` For example: `blade.fxtile.buy=Buy` For more information, see the document **Caplin Trader: Localization Overview And Concepts**.
Localization properties file	A file containing a set of **localization properties** for a given locale. Also called (when in context) a "**properties file**". Localization properties files are **UTF-8** encoded, with no **Byte-Order Mark**.
Localization support	The functions and features within a software application that allow it to (easily) support multiple **locales**. The software can then be localized to actually implement a given locale or locales (see **localization**).
Localization token	A marker embedded in the Caplin Trader software that indicates that an item, such as text, a date, or a number, must be localized. The marker takes the form `@{localization-key}`. When the software is converted to a given **locale** (see **localization**), the token is replaced with the corresponding text, date format, number format, and so on, for that locale. Each localization token identifies a **localization key**. Also called (when in context) a "**token**". For more information, see the document **Caplin Trader: Localization Overview And Concepts**.
Properties file	In this document, this term is short for **localization properties file**.
Token	In this document, this term is short for **localization token**.
Unicode	A computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. [Wikipedia contributors, "Unicode," *Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/wiki/Unicode (accessed April 6, 2011).] The Unicode standard is maintained by the Unicode Consortium (see http://unicode.org).
UTF-8	A way of encoding **Unicode** using multi-byte characters. The **localization properties files** used in Caplin Trader applications must be UTF-8 encoded.

Contact Us

Caplin Systems Ltd
Cutlers Court
115 Houndsditch
London EC3A 7BR
Telephone: +44 20 7826 9600
www.caplin.com