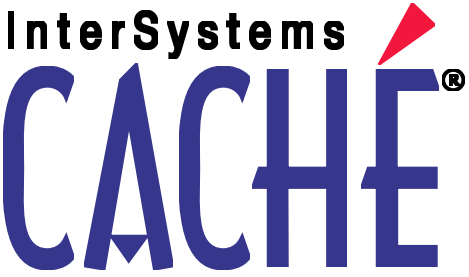

System Classes for National Language Support

Version 2009.1
30 June 2009

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

System Classes for National Language Support
Caché Version 2009.1 30 June 2009

This book was assembled and formatted in Adobe Page Description Format (PDF) using tools and information from the following sources: Sun Microsystems, RenderX, Inc., Adobe Systems, and the World Wide Web Consortium at www.w3c.org. The primary document development tools were special-purpose XML-processing applications built by InterSystems using Caché and Java.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Customer Support
Tel: +1 617 621-0700
Fax: +1 617 374-9391
Email: support@InterSystems.com

Table of Contents

System Classes for National Language Support

Modern applications are often designed so that they can adapt to various languages and regions without engineering changes. This process is called "internationalization." The act adapting an application to a specific region or language by adding specific components for that purpose is "localization."

The set of parameters that defines the user language, country and any other, special variant preferences is called a "locale." Locales specify the conventions for the input, output and processing of data. These are such things as

- Number formats
- Date and time formats
- Currency symbols
- The sort order of words
- Automatic translation of strings to another character set

A locale often is identified by noting the language in use and its geographic region (or other variation). These are usually given by the International Standards Organization (ISO) abbreviations for language and location. For example, "en-us" can represent the conventions of the English language as it is used in the United States, and "en-gb" as English is used in the United Kingdom (Great Britain). Similarly, "zh" is the abbreviation for Chinese with "zh-tw" representing traditional Chinese (as used on the island of Taiwan), "zh-cn" representing Simplified Chinese used in the People's Republic of China.

1 The %SYS.NLS Classes

Caché provides support for localization, referred to as National Language Support (or NLS), through the classes in the package, %SYS.NLS. These classes contain the information Caché needs to adapt an internationalized program to its runtime circumstances. Their functions are summarized in this section. Additional information is given with the class documentation for each class.

Note: Using any of these classes, an application can obtain the values currently set for the system or the process. Changing the values associated with the process takes effect immediately. Applications that wish to change the system settings must define a new locale with the appropriate values and direct Caché to start using the new locale.

1.1 %SYS.NLS.Locale

The properties in %SYS.NLS.Locale reflect the main characteristics of the current locale which a programmer might need to consult. These properties are merely informative. Changing any of them will not affect any behavior of the system.

1.2 %SYS.NLS.Device

The class, %SYS.NLS.Device, reflects some NLS properties for the current device, not necessarily the device that was current when the object was instantiated.

Usually, the properties for a specific device are set when the device is opened. This guarantees that the correct translations will be used. It is possible to change the translation table once the device is open by changing the XLTTable property in the process instance of this class, but this is not recommended without a solid reason for doing so.

Other useful properties in the %SYS.NLS.Device class allow more control over the treatment of error conditions that occur during translations. By default, when a character cannot be handled by the current table, no error is triggered and the offending character is translated as a question mark (?. This character, called the "replacement value" or "replacement string" can be changed to any other string. Furthermore, instead of silently translating undefined characters, it is possible to issue an error. This behavior is called the "default action," and the possible choices are:

- 0 = Generate error
- 1 = Replace the untranslatable character with the replacement value
- 2 = Ignore the error and pass the untranslatable character through

There are separate properties for the input and output operations in the properties of this class:

- *InpDefaultAction*
- *InpReplacementValue*
- *OutDefaultAction*
- *OutReplacementValue*

1.3 %SYS.NLS.Format

The class, %SYS.NLS.Format, contains a number of properties that affect the behavior of $ZDATE() and related functions. These properties are inherited from the values defined for the current locale but can be altered at the process level without affecting other users. The properties *DateSeparator* and *TimeSeparator*, for example, hold the characters that separate date and time items respectively.

The documentation for $ZDATE, $ZDATEH, and $FNUMBER describes the effect of changing these values.

1.3.1 Locale Property

The *Locale* property in class %SYS.NLS.Format allows control of how the "look" of various values appear in the current process. For example:

- Locale= "" means that the system default formats (usually US English) are in effect for the current process.
- Locale= "<locale_name>" (such as "rusw" or "csy8") means that the formats come from that locale.
- Locale= "Current" means use the locale of the system.

The property can be changed after the object is instantiated or by passing the desired locale to the **%New()** method as in

```
Set fmt = ##class(%SYS.NLS.Format).%New("jpnw")
```

These changes are visible only to the current process. They affect neither other processes nor the system default.

1.4 %SYS.NLS.Table

The class, %SYS.NLS.Table, can instantiate objects that reflect either the system default or the current process settings for the various categories of tables. A table is the basic NLS mechanism that allows application data to be accepted as input, ordered, and displayed in the format appropriate to the specified locale. As with %SYS.NLS.Locale, changing any property of a system object will not affect the system. However, changing a property from a process object will cause the associated behavior to change immediately.

NLS tables can be classified into I/O and Internal tables. Each table type has its own set of related data:

1.4.1 I/O Tables

These tables translate between the basic underlying character set supported by the current locale in which the systems is operating and a foreign character set supported by some entity outside Caché. The locale character set might be, for example, Latin2 (more properly known as ISO 8859-2) and the foreign character set might be UTF-8, generally used to communicate with the Caché Terminal. Thus, on output, a table like Latin2–to-UTF8 would be used and, on input, a reverse mapping table would be needed, UTF8–to-Latin2.

Although there are two tables involved here (one for input and another for output), these tables usually complement one another. For simplicity, when speaking of locale definitions and system defaults,

Caché uses a single name for a pair of I/O tables. This name is usually the name of the foreign character set, with the tacit assumption that the other half is the locale character set. However, when creating custom tables, any name that conveys the meaning of the exchange can be chosen.

I/O tables are used in "devices." Some of these, like "Caché processes" and "System call," are not really devices. However, like devices, they represent an interface between a Caché process and the external world (a callin/callout function or the operating system, respectively) where translation is also needed.

- Caché terminal
- Other terminal connections
- External files
- Magnetic tape
- TCP/IP connections
- Connections to DSM-DDP and DTM-DCP systems
- Printer
- Caché processes
- System call

Note: The term "device" is interpreted liberally to apply to those interfaces where Caché meets the external world. In that sense, it includes both the process and system call interfaces.

1.4.2 Internal Tables

The internal tables also map strings of characters from the current local character set to some other value, but they are not intended to be used in communication with the external world. The internal tables identify characters that are part of:

- Pattern matching

  Identify the characters that match certain pattern codes such as letters, numbers, punctuation, and so on.

- Identifiers

  Identifier tables indicate which characters can be used in identifiers.

- Uppercase, lowercase alphabets, and uppercase when used in titles

  These similar in structure to the I/O tables; they map from one character set to another which just happens to be the same set. However, they are used in the context of $ZCONVERT(), not with some I/O operation.

- Collation ordering

These tables map a string of characters into an internal representation of that string suitable for use in global subscripts. Different languages have differing rules about how words should collate in dictionary order; these rules are encapsulated in a collation table.

- $X/$Y action

  These tables map characters into values that indicate how they interact with the $X and $Y special variables. Should $X and/or $Y be incremented after this character is output? Is the character printable? These are questions that a $X/$Y table answers.

Note: The list of available collations in any version of Caché is fixed. Customers whose needs are not met by one of the existing collations are asked to contact the InterSystems Worldwide Support Center for assistance.

2 Examples Using %SYS.NLS

These examples are all executable but some do not have a "RunIt" button associated with them. This is because they manipulate process-default values for the current locale. If you wish to execute them, please run them in a separate process, such as the InterSystems Terminal facility (Windows), or via aTCP/IP connection.

2.1 Display Current Locale Information

This example displays information about the current system locale:

```
Set Info = ##class(%SYS.NLS.Locale).%New()
Set Items = "Name" _
            "/Description" _
            "/Country" _
            "/CountryAbbr" _
            "/Language" _
            "/LanguageAbbr" _
            "/Currency" _
            "/CharacterSet"

Write !
For i = 1 : 1 : $LENGTH(Items, "/")
{
  Set Item = $PIECE(Items, "/", i)
  Write $JUSTIFY(Item, 15),": ", $ZOBJProperty(Info, Item), !
}
```

2.2 Display Available Locales

This example displays information about the available locales:

```
Znspace "%SYS"
Set locales = ##class(%Library.ResultSet).%New("Config.NLS.Locales:List")
If $IsObject(locales) {
  Set locales.RuntimeMode = 1
  Set sc = locales.Execute("*")
  If $SYSTEM.Status.IsOK(sc) {
    Write !
    While locales.Next() {
      Write locales.Data("Name"), " - ", locales.Data("Description"), !
    }
  }
}
```

2.3 Display System and Process Table Data

This should display the same values for the system and process tables unless some properties have been externally altered before running this example.

```
Set IOTables = "Process" _
               "/CacheTerminal" _
               "/OtherTerminal" _
               "/File" _
               "/Magtape" _
               "/TCPIP" _
               "/DSMDDP" _
               "/DTMDCP" _
               "/SystemCall" _
               "/Printer"
Set IntTables = "PatternMatch" _
                "/Identifier" _
                "/Uppercase" _
                "/Lowercase" _
                "/Titlecase" _
                "/Collation" _
                "/XYAction"

// iterate over the systems, and then the process data
For Type = "System", "Process"
{
  Write !
  Set Table = ##class(%SYS.NLS.Table).%New(Type)
  Write "Type: ", Type, !

  Write "I/O Tables", !
  For i = 1 : 1 : $LENGTH(IOTables, "/")
  {
    Set PropName = $PIECE(IOTables, "/", i)
    Write $JUSTIFY(PropName, 15), ": ", $ZOBJPROPERTY(Table, PropName), !
  }

  Write "Internal Tables", !
  For i = 1 : 1 : $LENGTH(IntTables, "/")
  {
    Set PropName = $PIECE(IntTables, "/", i)
    Write $JUSTIFY(PropName, 15), ": ", $ZOBJPROPERTY(Table, PropName), !
  }
}
```

2.4 Changing Date and Time Displays

The %SYS.NLS.Format class contains properties *DateSeparator* and *TimeSeparator*, for example, hold the characters used to separate the components of date and time items respectively. In the United States default locale, "enu8" (or "enuw" for Unicode systems), these are the characters "/" and ":", respectively. The following example shows how these may be altered:

```
// display the current defaults
// date is 10 April 2005
// time is 6 minutes 40 seconds after 11 in the morning
Write $ZDATE("60000,40000"), !

// now change the separators and display it again
Set fmt = ##class(%SYS.NLS.Format).%New()
Set fmt.DateSeparator = "_"
Set fmt.TimeSeparator = "^"
Write !, $ZDATE("60000,40000")
```

This example changes the month names to successive letters of the ASCII character set. It is easily done by setting property *MonthName* to a list of the month names. Each month must be preceded by a blank:

```
// get the format class instance
Set fmt = ##class(%SYS.NLS.Format).%New()

// define the month names
Set Names = " AAA BBB CCC DDD EEE FFF GGG HHH III JJJ KKK LLL"
Set fmt.MonthName = Names

// show the result
Write $ZDATE("60000,40000")
```

2.5 Changing the Way Numbers Are Displayed

In addition to date and time, some properties in %SYS.NLS.Format control the way that numbers are interpreted by $Number(). In English locales, the decimal point is used separate the integer from the fractional part of a number; a comma to separate groups of 3 digits. This too can be altered:

```
// give the baseline display
Write $Number("123,456.78"), !

Set fmt = ##class(%SYS.NLS.Format).%New()
// use "/" for groups of digits
Set fmt.NumericGroupSeparator = "."

// group digits in blocks of 4
Set fmt.NumericGroupSize = 4

// use ":" for separating integer and fractional parts
Set fmt.DecimalSeparator = ","

// try interpreting again
Write $Number("12.3456,78"), !
```

2.6 Setting the Translation for a File

The following shows that an application can control the representation of data written to a file.

```
// show the process default translation to UTF-8
Set Tbl = ##class(%SYS.NLS.Table).%New("Process")
Write "Process default translation: ", Tbl.File, !

// create and open a temporary file
// use XML for the translation
Set TempName = ##class(%Library.File).TempFilename("log")
Set TempFile = ##class(%Library.File).%New(TempName)
Do TempFile.Open("WSNK\XML\")
Write "Temp file: ", TempFile.CanonicalNameGet(), !

// write a few characters to show the translation
// then close it
Do TempFile.WriteLine(("--" _ $CHAR(38) _ "--"))
Do TempFile.Close()

// now re-open it in raw mode and show content
Do TempFile.Open("RSK\RAW\")
Do TempFile.Rewind()
Set MaxChars = 50
Set Line = TempFile.Read(.MaxChars)
Write "Contents: """, Line, """", !

// finish
Do TempFile.Close()
Do ##class(%Library.File).Delete(TempName)
Set TempFile = ""
```

3 The Config.NLS Classes

Unlike %SYS.NLS, which is available everywhere and is intended for general use, the classes in Config.NLS require that the application be run in namespace %SYS and have administrative privileges. Normally, administrators that need to create custom locales and tables would use the NLS pages in the Management Portal. Alternatively, the character-mode utility ^NLS can be used for this purpose. Only users with very special requirements should need to use Config.NLS.

There are three classes in package Config.NLS:

- Locales – contain all the definitions and defaults for a country or geographical region.
- Tables – contain a high level description of tables, but not the mapping itself.
- SubTables – Contain the character mappings proper and may be shared by more than one Table.

The main reason for having separate Tables and SubTables classes is to avoid duplication of data. It is possible to have Tables for different character sets that happen to share the same mappings and thus the same SubTable. Also, Tables define a default action and a replacement value (see description of these properties in %SYS.NLS above). Therefore, it is possible to have separate Tables in which these

attributes are different even though they share the same SubTable. This flexibility adds some complexity in managing the correct relationships between Tables and SubTables, but the gains make it worthwile. The separation of Tables from SubTables is kept hidden from users in the Management Portal, ^NLS and in the %SYS.NLS classes, where all the housekeeping is done. However, when working with Config.NLS this needs to be done explicitly.

3.1 Conventions for Naming User-Defined Locales and Tables

In order to differentiate between tables provided with Caché and custom ones created by users, InterSystems has adopted theconvention that the name of the user–created encoding should begin with "y," such as, "XLT-yEBCDIC-Latin1" and "XLT-Latin1-yEBCDIC."

When a custom SubTable is created from a copy of some InterSystems SubTable, the utilities that perform this task automatically use the same name and append a numeric suffix. Thus, copies of the Latin2<->Unicode SubTablewould be named "XLT-Latin2-Unicode.0001" and "XLT-Unicode-Latin2.0001" (or use some other non-conflicting numeric suffix).

CAUTION: User-defined tables, sub-tables and locales that do not follow this convention may be deleted during a system upgrade. The way to avoid this is to export user-defined tables and locales to XML files and re-import them after the upgrade.

4 Examples Using Config.NLS

4.1 Listing the Available Locales

This example uses a pre-defined query to obtain and list the available locale identifiers and descriptions. At the time Caché is installed, only the locales appropriate to the system – 8-bit locales for systems that only support 8-bit characters; Unicode locales for systems that support multibyte characters – are made available for use.

```
// use the query in Config.NLS to get the locales
ZNspace "%SYS"
Set Query = ##class(%Library.ResultSet).%New("Config.NLS.Locales:List")
Set code = Query.Execute("*")
If (##class(%SYSTEM.Status).IsError(code))
{
  Do ##class(%SYSTEM.Status).DisplayError(Code)
  Quit
}

// display each of them in turn
Write "Available locales and descriptions", !
While (Query.Next(.code))
{
  If (##class(%SYSTEM.Status).IsError(code))
  {
    Do ##class(%SYSTEM.Status).DisplayError(Code)
```

```
      Quit
    }
    Write Query.Get("Name"), ": ", Query.Get("Description"), !
  }
```

4.2 Listing the Tables in a Specific Locale

The following example shows the tables that make up the Unicode locale for United States English (if it is available).

```
ZNspace "%SYS"

// establish the locale identifier, try
// United States - English - Unicode
// United States - English - 8-bit
Set Loc = "enuw"
Do ##class(Config.NLS.Locales).Exists(Loc, .Ref, .Code)
If (##class(%SYSTEM.Status).IsError(Code))
{
  Set Loc = "enu8"
  Do ##class(Config.NLS.Locales).Exists(Loc, .Ref, .Code)
  If (##class(%SYSTEM.Status).IsError(Code))
  {
    Do ##class(%SYSTEM.Status).DisplayError(Code)
    Quit
  }
}

// get the local array of table names
Write "Tables for locale: ", Loc, !
Do Ref.GetTables(.Tables)
Set Type = $ORDER(Tables(""))
While (Type '= "")
{
  Set Name = $ORDER(Tables(Type, ""))
  While (Name '= "")
  {
    Set Mod = $ORDER(Tables(Type, Name, ""))
    While (Mod '= "")
    {
      Write Type, " - ", Name, " - ", Mod, !
      Set Mod = $ORDER(Tables(Type, Name, Mod))
    }
    Set Name = $ORDER(Tables(Type, Name))
  }
  Set Type = $ORDER(Tables(Type))
}
```

4.3 Creating a Custom Locale

This example will provide a template for creating a custom locale with a custom table. The custom table will translate between EBCDIC (the common form used in the US) and Latin-1 (ISO-8859–1). For more details, please see the documentation for the respective classes.

As for any other table, first we need to get the definition for the character mappings. For this example we are using the data file from the site, International Components for Unicode. The relevant data file is a text file with comment lines starting with “#’ and then a series of translation definition lines of the form:

```
<Uuuuu>  \xee |0
```

A small excerpt of the file looks like:

```
#
#UNICODE EBCDIC_US
#_______ _________
<U0000>  \x00 |0
<U0001>  \x01 |0
<U0002>  \x02 |0
<U0003>  \x03 |0
<U0004>  \x37 |0
<U0005>  \x2D |0
...
```

The lines indicate that Unicode character Uaaaa maps to EBCDIC character \xbb (where aaaa and bb are expressed in hexadecimal). We assume that the table is reversible and that EBCDIC character \xbb maps back to Unicode character Uaaaa. This allows us to create both sides (i.e., EBCDIC->Latin1 and Latin1->EBCDIC) from the same data file in a single scan. Since the Unicode range is just from 0 to 255, this is actually a Latin-1 table.

The process proceeds by first creating the SubTable, then the Table, and finally the Locale. For the first step, the process creates two SubTables objects, initializes their Name and Type properties, and then fills in the FromTo mapping array with data read from the definition file.

SubTable names take the form, Type–FromEncoding–ToEncoding. The Type for regular I/O translations is "XLT" and so the SubTable names will be "XLT-yEBCDIC-Latin1" and "XLT-yLatin1-EBCDIC."

The following code creates the SubTable objects. In a real world program, the code would perform a number of consistency checks that omitted here for the sake of clarity. This example deletes an existing previous versions of the same objects (SubTables, Tables and Locales) so that it can run the example multiple times. More properly, the application should check for the existence of previous objects using the class method Exists() and take a different action if they are already present.

```
// Names for the new SubTables (save for later)
Set nam1 = "XLT-Latin1-yEBCDIC"
Set nam2 = "XLT-yEBCDIC-Latin1"

// Delete existing SubTables instances with same ids
Do ##class(Config.NLS.SubTables).Delete(nam1)
Do ##class(Config.NLS.SubTables).Delete(nam2)

// Create two SubTable objects
Set sub1 = ##class(Config.NLS.SubTables).%New()
Set sub2 = ##class(Config.NLS.SubTables).%New()

// Set Name and Description
Set sub1.Name = nam1
Set sub1.Description = "ICU Latin-1->EBCDIC sub-table"
Set sub2.Name = nam2
Set sub2.Description = "ICU EBCDIC ->Latin-1 sub-table"
```

The SubTable contains a property, *type*, that is a small integer indicating whether we are dealing with a multi-byte translation or not. This example sets type to zero indicating a single-byte mapping. The mapping is initialized so that code points (characters) not defined in the data file are mapped to themselves.

```
  // Set Type (single-to-single)
  Set sub1.Type = 0
  Set sub2.Type = 0

  // Initialize FromTo arrays
  For i = 0 : 1 : 255
  {
    Do sub1.FromTo.SetAt(i, i)
    Do sub2.FromTo.SetAt(i, i)
  }
```

Next the application reads the file. Definitions in the file override those set as the default mapping. The function, $ZHEX(), converts the codes from hexadecimal to decimal.

```
  // Assume file is in the mgr directory
  Set file = "glibc-EBCDIC_US-2.1.2.ucm"

  // Set EOF exit trap
  Set $ZTRAP = "EOF"

  // Make that file the default device
  Open file
  Use file
  For
  {
    Read x
    If x?1"<U"4AN1">".E
    {
      Set uni = $ZHEX($E(x,3,6)),ebcdic = $ZHEX($E(x,12,13))
      Do sub1.FromTo.SetAt(ebcdic,uni)
      Do sub2.FromTo.SetAt(uni,ebcdic)
     }
  }

EOF  // No further data
  Set $ZT = ""
  Close file

  // Save SubTable objects
  Do sub1.%Save()
  Do sub2.%Save()
```

The character mappings are now complete. The next step is to create the Table objects that reference the SubTables just defined. Table objects are really descriptors for the SubTables and have only a few properties. The following code makes the connection between the two:

```
  // Delete existing Tables instances with same ids
  Do ##class(Config.NLS.SubTablels).Delete("XLT", "Latin1", "yEBCDIC")
  Do ##class(Config.NLS.SubTablels).Delete("XLT", "yEBCDIC", "Latin1")

  // Create two Table objects
  Set tab1 = ##class(Config.NLS.Tables).%New()
  Set tab2 = ##class(Config.NLS.Tables).%New()

  // Set description
  Set tab1.Description = "ICU loaded Latin-1 -> EBCDIC table"
  Set tab2.Description = "ICU generated EBCDIC -> Latin-1 table"

  // Set From/To encodings
  Set tab1.NameFrom = "Latin1"
  Set tab1.NameTo = "yEBCDIC"
  Set tab2.NameFrom = "yEBCDIC"
  Set tab2.NameTo = "Latin1"
```

```
// Set SubTable
Set tab1.SubTableName = nam1
Set tab2.SubTableName = nam2

// Set Type
Set tab1.Type = "XLT"
Set tab2.Type = "XLT"

// Set Default Action
// 1 = Replace with replacement value
Set tab1.XLTDefaultAction = 1
Set tab2.XLTDefaultAction = 1

// Set Replacement value of "?"
Set tab1.XLTReplacementValue = $ASCII("?")
Set tab2.XLTReplacementValue = $ASCII("?")

// Set Reversibility
// 1 = Reversible
// 2 = Generated
Set tab1.XLTReversibility = 1
Set tab2.XLTReversibility = 2

// Set Translation Type
// 0 = non-modal to non-modal
Set tab1.XLTType = 0
Set tab2.XLTType = 0

// Save Table objects
Do tab1.%Save()
Do tab2.%Save()
```

With the Tables defined, the last step of the construction is to define a locale object that will incorporate the new tables. The application creates an empty Locale object and fills in each of the properties as was done for the Tables and SubTables. A Locale, however, is bigger and more complex. The easiest way to make a simple change like this is to copy an existing locale and change only what we need. This process uses "enu8" as the source locale and names the new one, "yen8." The initial "y" makes it clear this is a custom locale and should not be deleted on upgrades.

```
// Delete existing Locales instance with the same id
Do ##class(Config.NLS.Locales).Delete("yen8")

// Open source locale
Set oldloc = ##class(Config.NLS.Locales).%OpenId("enu8")

// Create clone
Set newloc = oldloc.%ConstructClone()

// Set new Name and Description
Set newloc.Name = "yen8"
Set newloc.Description = "New locale with EBCDIC table"
```

With the locale in place, the process now to adds the EBCDIC table to be list of I/O tables that are loaded at startup. This is done by creating a new node in the array, *XLTTables*, specifically

```
XLTTables(<TableName>) = <components>
```

At the locale level, a single name identifies the pair of Input and Output tables. Since the name does not need to start with "y," the *TableName* will be "EBCDIC." The *component* is a 4-element list with the following values:

1. The input “From” encoding
2. The input “To” encoding
3. The output “From” encoding
4. The output “To” encoding

The following code adds the table to the list of available locales:

```
// Add new table to locale
Set component = $LISTBUILD("yEBCDIC", "Latin1", "Latin1", "yEBCDIC")
Do newloc.XLTTables.SetAt(component, "EBCDIC")
```

If this locale will be frequently used, for example, for reading with EBCDIC magnetic tapes, the following code will set it as the default for this class of devices:

```
// Set default for Magnetic Tapes
Set newloc.TranMagTape = "EBCDIC"

// Save the changes
Do newloc.%Save()
```

Before the locale is usable by Caché, it must be compiled into its internal form.. This is also sometimes called validating the locale. The **IsValid()** class method does a detailed analysis and returns two arrays, one for errors and one for warnings, with human-readable messages if the locale is not properly defined.

```
// Check locale consistency
If '##class(Config.NLS.Locales).IsValid("yen8", .Errors, .Warns)
{
  Write !,"Errors: "
  ZWrite Errors
  Write !,"Warnings: "
  ZWrite Warns
  Quit
}

// Compile new locale
Set status = ##class(Config.NLS.Locales).Compile("yen8")
If (##class(%SYSTEM.Status).IsError(status))
{
  Do $System.OBJ.DisplayError(status)
}
Else
{
  Write !,"Locale yen8 successfully created."
}
```

5 Using %Library.GlobalEdit to Set Collation for a Global

Since Cache 4.1, the collation of newly created globals is automatically set to the default collation of the database where the global is created. When a database is created, you can set the default collation

of the database — either Caché Standard or one of the collations installed on the system. Once the default collation of the database is set, any globals created in this database are created with this default collation.

Because any globals created in a database have the database's default collation, Caché also supports the ability to override this behavior. To do this, use the **Create** method in the class %Library.GlobalEdit supplying the collation desired:

```
Set sc = ##class(%Library.GlobalEdit).Create(ns,
                                             global,
                                             collation,
                                             growthblk,
                                             ptrblock,
                                             keep,
                                             journal,
                                             .exists)
```

where:

- *ns* — Specifies the namespace, where `""` indicates the current namespace, or `^^directoryname` references a specific directory.
- *global* — Specifies the global name, including leading ^, such as ^cz*2*.
- *collation* — Specifies the collation, where collation is one of the supported collations.
- *growthblk* — Specifies the starting block for data.
- *ptrblk* — Specifies the starting block for pointers.
- *keep* — Specifies whether or not to keep the global's directory entry when the global is killed. Setting this to 1 preserves the collation, protection, and journal attributes if the global is killed.
- *journal* — This argument is no longer relevant and is ignored.
- *exists* — Specifies, by reference, a varaible whose return value indicates whether the global already exists.

In environments where some globals require different collations from other globals, InterSystems recommends that you set up a database for each different collation, and that you add a global mapping to the namespace to map each global to the database with its required collation. This method allows mixed collations to be used without changing application code to specially use the **Create** method call.

Note: In older versions of Caché, a newly created global would have the default collation of the process creating it. This method of setting the collation is no longer supported.

5.1 Supported Collations

The following are supported in Caché, for use in the collation argument of the **CreateGlobal^%DM** routine:

- 5 — Cache standard
- 10 — German1
- 11 — Portuguese1
- 12 — Polish1
- 13 — German2
- 14 — Spanish1
- 15 — Danish1
- 17 — Greek1
- 18 — Czech1
- 19 — Czech2
- 20 — Portuguese2
- 21 — Finnish1
- 22 — Japanese1
- 23 — Cyrillic2
- 24 — Polish2
- 26 — Chinese2