Using Perl with Caché

Version 2018.1
2020-11-13

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

Using Perl with Caché
Caché Version 2018.1 2020-11-13

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book is a guide to the Caché Perl Language Binding.

This book contains the following sections:

- The Caché Perl Binding
- Using the Perl Binding
- Perl Client Class Reference

There is also a detailed Table of Contents.

For general information, see *Using InterSystems Documentation*.

1

The Caché Perl Binding

The Caché Perl binding provides a simple, direct way to manipulate Caché objects from within a Perl application. It allows Perl programs to establish a connection to a database on Caché, create and open objects in the database, manipulate object properties, save objects, run methods on objects, and run queries. All Caché datatypes are supported.

The Perl binding offers complete support for object database persistence, including concurrency and transaction control. In addition, there is a sophisticated data caching scheme to minimize network traffic when the Caché server and the Perl applications are located on separate machines.

This document assumes a prior understanding of Perl and the standard Perl modules. Caché does not include a Perl interpreter or development environment.

1.1 Perl Binding Architecture

The Caché Perl binding gives Perl applications a way to interoperate with objects contained within a Caché server. The Perl binding consists of the following components:

- *The Intersys::PERLBIND module* — a Perl C extension that provides your Perl application with transparent connectivity to the objects stored in the Caché database.
- *The Caché Object Server* — a high-performance server process that manages communication between Perl clients and a Caché database server. It communicates using standard networking protocols (TCP/IP), and can run on any platform supported by Caché. The Caché Object Server is used by all Caché language bindings, including Perl, Python, C++, Java, JDBC, and ODBC.

The basic mechanism works as follows:

- You define one or more classes within Caché. These classes can represent persistent objects stored within the Caché database or transient objects that run within a Caché server.
- At runtime, your Perl application connects to a Caché server. It can then access instances of objects within the Caché server. Caché automatically manages all communications as well as client-side data caching. The runtime architecture consists of the following:
  - A Caché database server (or servers).
  - The Perl interpreter (see Perl Client Requirements).
  - A Perl application. At runtime, the Perl application connects to Caché using either an object connection interface or a standard ODBC interface. All communications between the Perl application and the Caché server use the TCP/IP protocol.

1.2 Quick Start

Here are examples of a few basic functions that make up the core of the Perl binding:

- *Create a connection and get a database*

```
$url = "localhost[1972]:Samples"
$conn = Intersys::PERLBIND::Connection->new($url,"_SYSTEM","SYS",0);
$database = Intersys::PERLBIND::Database->new($conn);
```

  `$database` is your logical connection to the namespace specified in `$url`.

- *Open an existing object*

```
$person = $database->openid("Sample.Person","1", -1, -1);
```

  `$person` is your logical connection to a `Sample.Person` object on the Caché server.

- *Create a new object*

```
$person = $database->create_new("Sample.Person",undef);
```

- *Set or get a property*

```
$person->set("Name","Adler, Mortimer");
$name = $person->get("Name");
```

- *Run a method*

```
$answer = $person->Addition(17,20);
```

- *Save an object*

```
$person->run_obj_method("%Save")
```

- *Get the id of a saved object*

```
$id = $person->run_obj_method("%Id")
```

- *Run a query*

```
$query = $database->alloc_query();
$query->prepare("SELECT * FROM SAMPLE.PERSON WHERE ID=?",$sqlcode);
$query->set_par(1,2);
$query->execute($sqlcode);
while (@data_row = $query->fetch($sqlcode)) {
   $colnum = 1;
   foreach $col (@data_row) {
      $col_name = $query->col_name($colnum);
      print "column name = $col_name, data=$col\n";
   $colnum++; }
}
```

1.3 Installation and Configuration

The standard Caché installation places all files required for Caché Perl binding in <cachesys>/dev/perl. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*). You should be able to run any of the Perl sample programs after performing the following installation procedures.

1.3.1 Perl Client Requirements

Caché provides client-side Perl support through the Intersys::PERLBIND module, which implements the connection and caching mechanisms required to communicate with a Caché server.

This module requires the following environment:

- Perl version 5.10 — Intersystems supports only the ActiveState distribution (www.activestate.com).
- A C++ compiler — On Windows, use Visual Studio 2008 or 2010 (for more information, see Windows Compilers for Perl Modules on the ActiveState site). On UNIX®, use GCC.
- Your PATH must include the <cachesys>\bin directory. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*).
- Your environment variables must be set to support C compilation and linking (for example, on Windows call VSVARS32.BAT). Otherwise, linker errors will be reported on the make step.

1.3.2 UNIX® Installation

- Make sure <cachesys>/bin is on your PATH and in your LD_LIBRARY_PATH (or DYLD_LIBRARY_PATH on OSX). For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*.

  For example:

  ```
  export PATH=/usr/cachesys/bin:$PATH
  export LD_LIBRARY_PATH=/usr/cachesys/bin:$LD_LIBRARY_PATH
  ```

- Run Makefile.PL (located in <cachesys>/dev/perl). You can supply the location of <cachesys>as an argument. For example:

  ```
  perl Makefile.PL /usr/cachesys
  ```

  If you run it without an argument, it will prompt you for the location of <cachesys>.

- After making sure that <cachesys>/bin is on your path, run:

  ```
  make
  ```

- If make runs with no errors, test it with:

  ```
  make test
  ```

- If make test is successful, run:

  ```
  make install
  ```

1.3.3 Windows Installation

- Make sure <cachesys>\bin is on your PATH. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*).
- Run Makefile.PL (located in <cachesys>\dev\perl). You can supply the location of <cachesys> as an argument. For example:

  ```
  perl Makefile.PL C:\Intersystems\Cache
  ```

  If you run it without an argument, it will prompt you for the location of <cachesys>.

- After making sure that <cachesys>\bin is on your path, run:

```
nmake
```

- If nmake runs with no errors, test it with:

```
nmake test
```

- If nmake test is successful, run:

```
nmake install
```

1.3.4 Caché Server Configuration

Very little configuration is required to use a Perl client with a Caché server. The Perl sample programs provided with Caché should work with no change following a default Caché installation. This section describes the server settings that are relevant to Perl and how to change them.

Every Perl client that wishes to connect to a Caché server needs the following information:

- A URL that provides the server IP address, port number, and Caché namespace.
- A username and password.

By default, the Perl sample programs use the following connection information:

- connection string: `"localhost[1972]:Samples"`
- username: `"_SYSTEM"`
- password: `"SYS"`

Check the following points if you have any problems:

- Make sure that the Caché server is installed and running.
- Make sure that you know the IP address of the machine on which the Caché server is running. The Perl sample programs use `"localhost"`. If you want a sample program to default to a different system you will need to change the connection string in the code.
- Make sure that you know the TCP/IP port number on which the Caché server is listening. The Perl sample programs use `"1972"`. If you want a sample program to default to a different port, you will need change the number in the sample code.
- Make sure that you have a valid username and password to use to establish a connection. (You can manage usernames and passwords using the Management Portal). The Perl sample programs use the administrator username `"_SYSTEM"` and the default password `"SYS"` or `"sys"`. Typically, you will change the default password after installing the server. If you want a sample program to default to a different username and password, you will need to change the sample code.
- Make sure that your connection URL includes a valid Caché namespace. This should be the namespace containing the classes and data your program uses. The Perl samples connect to the SAMPLES namespace, which is pre-installed with Caché.

1.4 Sample Programs

Caché comes with a set of sample programs that demonstrate the use of the Caché Perl binding. These samples are located in the <cachesys>/dev/perl/samples/ subdirectory of the Caché installation. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*). They are named, numbered, and implemented to correspond to the Java binding samples.

The sample programs include:

- CPTest2.pl — Get and set properties of an instance of Sample.Person.
- CPTest3.pl — Get properties of embedded object Sample.Person.Home.
- CPTest4.pl — Update embedded object Sample.Person.Home.
- CPTest5.pl — Process datatype collections.
- CPTest6.pl — Process the result set of a ByName query.
- CPTest7.pl — Process the result set of a dynamic SQL query.
- CPTest8.pl — Process employee subclass and company/employee relationship.

All of these applications use classes from the Sample package in the SAMPLES namespace (accessible in Atelier). If you have not used the Sample package before, you should open it in Atelier and make sure it is properly compiled.

Arguments

The sample programs are controlled by various switches that can be entered as arguments to the program on the command line. The sample program supplies a default value if you do not enter an argument.

For example, CPTest2.pl accepts the following optional arguments:

- `-user` — the username you want to login under (default is `"_SYSTEM"`).
- `-password` — the password you want to use (default is `"SYS"`).
- `-host` — the host computer to connect to (default is `"localhost"`).
- `-port` — the port to use (default is `"1972"`).

A `-user` argument would be specified as follows:

```
perl CPTest2.pl -user _MYUSERNAME
```

The CPTest7.pl sample accepts a `-query` argument that is passed to an SQL query:

```
perl CPTest7.pl -query A
```

This query will list all Sample.Person records containing names that start with the letter A.

2

Using the Perl Binding

This chapter provides concrete examples of Perl code that uses the Caché Perl binding. The following subjects are discussed:

- Perl Binding Basics — the basics of accessing and manipulating Caché database objects.
- Using Tied Variables — some Perl-specific ways to access Perl bindings.
- Using Collections — iterating through Caché lists and arrays.
- Using Relationships — manipulating embedded objects.
- Using Queries — running Caché queries and dynamic SQL queries.
- Using %Binary Data — moving data between Caché %Binary and Perl array of ord.
- Handling Exceptions — handling Perl exceptions and error messages from the Perl binding.
- Perl Binding Constraints and Limitations — constraints imposed by differences in the way that Perl and Caché handle certain datatypes

Many of the examples presented here are taken from the sample programs in <cachesys>/dev/perl/samples/ (see Default Caché Installation Directory in the *Caché Installation Guide* for the location of <cachesys> on your system). The argument processing and error trapping statements (try/catch) have been removed to simplify the code. See Sample Programs for details about loading and running the complete sample programs.

2.1 Perl Binding Basics

A Caché Perl binding application can be quite simple. Here is a complete sample program:

```
use Intersys::PERLBIND;
eval {
# Connect to the Cache' database
   $url = "localhost[1972]:Samples";
   $user = "_SYSTEM";
   $password = "SYS";
   $conn = Intersys::PERLBIND::Connection->new($url,$user,$password,0);
   $database = Intersys::PERLBIND::Database->new($conn);

# Create and use a Cache' object
   $person = $database->create_new("Sample.Person", undef);
   $person->set("Name","Doe, Joe A");
   print "Name: ",$person->get("Name"), "\n";
};
```

This code imports the Intersys::PERLBIND module, and then performs the following actions:

- Connects to the Samples namespace in the Caché database:

  - Defines the information needed to connect to the Caché database.
  - Creates a Connection object (`$conn`).
  - Uses the Connection object to create a Database object (`$database`).

- Creates and uses a Caché object:
  - Uses the Database object to create an instance of the Caché Sample.Person class.
  - Sets the Name property of the Sample.Person object.
  - Gets and prints the Name property.

The following sections discuss these basic actions in more detail.

2.1.1 Connecting to the Caché Database

The basic procedure for creating a connection to a namespace in a Cache database is as follows:

- Establish the physical connection:

```
$conn = Intersys::PERLBIND::Connection->new($url,$userid,$password,$timeout);
```

  The **Connection->new()** method creates a physical connection to a namespace in a Caché database. The `$url` parameter defines which server and namespace the Connection object will access. The Connection class also provides the **new_secure()** method for establishing secure connections using Kerberos. See Connections for a detailed discussion of both methods.

- Create a logical connection:

```
$database = Intersys::PERLBIND::Database->new($conn);
```

  The Connection object is used to create a Database object, which is a logical connection that lets you use the classes in the namespace to manipulate the database.

  The following code establishes a connection to the Samples namespace:

```
$address ="localhost"; # server TCP/IP address ("localhost" is 127.0.0.1)
$port = "1972"; # server TCP/IP port number
$namespace = "SAMPLES"; # sample namespace installed with Cache'

$url = $address."[$port]:".$namespace;
$user = "_SYSTEM";
$password = "SYS";

$conn = Intersys::PERLBIND::Connection->new($url,$user,$password,0);
$database = Intersys::PERLBIND::Database->new($conn);
```

2.1.2 Using Caché Database Methods

The **Intersys::PERLBIND::Database** class allows you to run Caché class methods and connect to Caché objects on the server. Here are the basic operations that can be performed with the Database class methods:

- *Create Objects*

  The **create_new()** method is used to create a new Caché object. The syntax is:

```
$object = $database->create_new($class_name, $initial_value)
```

  where `$class_name` is the name of a Caché class in the namespace accessed by `$database`. For example, the following statement creates a new instance of the Sample.Person class:

```
$person = $database->create_new("Sample.Person", undef);
```

In this example, the initial value of `$person` is undefined.

- *Open Objects*

  The **openid()** method is used to open an existing Caché object. The syntax is:

  ```
  $object = $database->openid($class_name, $id, $concurrency, $timeout)
  ```

  For example, the following statement opens the Sample.Person object that has an id with a value of 1:

  ```
  $person = $database->openid("Sample.Person", "1", -1, 0);
  ```

  Concurrency and timeout are set to their default values.

- *Run Class Methods*

  You can run class methods using the following syntax:

  ```
  $result = $database->run_class_method($classname,$methodname,LIST)
  ```

  where `LIST` is a list of method arguments. For example, the `$database->openid()` example shown previously is equivalent to the following code:

  ```
  $person = $database->run_class_method("Sample.Person","%OpenId","1");
  ```

  This method is the analogous to the Caché `##class` syntax. For example, the following code:

  ```
  $list = $database->run_class_method("%ListOfDataTypes","%New",undef);
  $list->Insert("blue");
  ```

  is exactly equivalent to the following ObjectScript code:

  ```
  set list=##class(%ListOfDataTypes).%New()
  do list.Insert("blue")
  ```

2.1.3 Using Caché Object Methods

The **Intersys::PERLBIND::Object** class provides access to Caché objects. Here are the basic operations that can be performed with the Object class methods:

- *Get and Set Properties*

  Properties are accessed through the **get()** and **set()** accessor methods. The syntax for these methods is:

  ```
  $value = $object->get($propname)
  $object->set($propname,$value)
  ```

  For example:

  ```
  $person->set("Name","Adler, Mortimer");
  $name = $person->get("Name");
  ```

  Private and multidimensional properties are not accessible through the Perl binding.

- *Run Object Methods*

  You can run object methods by calling them directly::

  ```
  $answer = $object->MethodName(LIST);
  ```

  For example:

  ```
  $answer = $person->Addition(17,20);
  ```

You can also use **run_obj_method()** to do the same thing:

```
$answer = $object->run_obj_method($MethodName,LIST);
```

For example:

```
$answer = $object->run_obj_method("Addition", 17, 20);
```

This method is useful when calling inherited methods (such as **%Save** or **%Id**) that are not directly available.

- *Save Objects*

  To save an object, use **run_obj_method()** to call **%Save**:

  ```
  $object->run_obj_method("%Save")
  ```

  To get the id of a saved object, use **run_obj_method()** to call **%Id**:

  ```
  $id = $object->run_obj_method("%Id")
  ```

- *Get Information about Methods and Properties*

  The Object class provides several methods that return information about the methods and properties in an object. The **get_methods()** and **get_properties()** methods return either the number of available items or a list of valid item names, depending on the context:

  ```
  $methodcount = $object->get_methods(); # scalar context
  @methodlist = $object->get_methods(); # list context

  $propcount = $object->get_properties(); # scalar context
  @proplist = $object->get_properties(); # list context
  ```

  The **is_method()** and **is_property()** methods test a string to see if it is a valid item name:

  ```
  $bool = $object->is_method($methodname);
  $bool = $object->is_property($propname);
  ```

2.2 Using Tied Variables

It is possible to tie an object returned by the Perl binding to a hash, and then access the properties of the object through the hash. For example:

```
   $object = $database->openid("Sample.Person", "1", -1, 0);
   $person = tie %person,"Intersys::PERLBIND::ObjectHash",( _object => $object)
   $name = $person{"Name"};
# The following line is equivalent to $person->set("Name","Tregar, Sam");
   $person{"Name"} = "Tregar, Sam";
```

Instead of getting the the name of person through `$person->get("Name")`, you can use the tie to get it through the hash, `$person{"Name"}`.

You can use the return value from the tie to run methods or you can use the "tied" built-in of Perl to run methods, for example:

```
$ans = $person->Addition(12,17);
print "ans=$ans\n";

$ans = tied(%person)->Addition(12,17);
print "ans=$ans\n";
```

In both cases, the value 29 is returned.

You can also use the tie to iterate through a list of properties:

```
while (($property, $value ) = each(%person)) {
   print "propety=$property value=$value\n";
}
```

Here is a complete example:

```
$database = Intersys::PERLBIND::Database->new($conn);
$personobj = $database->openid("Sample.Person", "1", -1, 0);

$person = tie %person,
   "Intersys::PERLBIND::ObjectHash",( _object => $personobj);
while (($propname, $value) = each(%person)) {
   print "property $propname = $value\n";
}
```

This example produces the following output, displaying each property and its value:

```
property Age = 61
property DOB = 1942-01-18
property SSN = 295-62-8728
property Home = Intersys::PERLBIND::Object=SCALAR(0x1831ee4)
property Name = Adler, Mortimer
property Office = Intersys::PERLBIND::Object=SCALAR(0x183eed0)
property Spouse =
property FavoriteColors = Intersys::PERLBIND::Object=SCALAR(0x1831e3c)
```

2.3 Using Collections and Lists

Caché %Collection objects are handled like any other Perl binding object. Caché %List variables are mapped to Perl array references. The following sections demonstrate how to use both of these items.

2.3.1 %Collection Objects

Collections are manipulated through object methods of the Caché %Collection class. The following example shows how you might manipulate a Caché %ListOfDataTypes collection:

```
# Create a %ListOfDataTypes object and add a list of colors
   $newcolors = $database->run_class_method("%ListOfDataTypes","%New",undef);
   @color_list = qw(red blue green);
   for $color (@color_list) {
      $newcolors->Insert($color);
      print " added >$color<\n";
   };

# Add the list to a Sample.Person object
   $person = $database->openid("Sample.Person",$id,-1,0);
   $person->set("FavoriteColors",$newcolors);

# Get the list back from $person and print it out.
   $colors = $person->get("FavoriteColors");
   print "Number of colors: ",$colors->get("Size"), "\n";
   do {
      $color = $colors->GetNext($i);
      print " New Color #$i = $color\n" if defined($i);
   } until (!defined($i));

# Remove and replace the second element
   $colors->RemoveAt(2) if $colors->get("Size") > 0;
   $colors->InsertAt("purple",2);

# Show the changes to the collection
   print("Modified 'FavoriteColors' list:\n");
   do {
      $color = $colors->GetNext($i);
      print " Element #$i = $color\n" if defined($i);
   } until (!defined($i));
```

2.3.2 %List Variables

The Perl binding maps Caché %List variables to Perl array references.

CAUTION: While a Perl array has no size limit, Caché %List variables are limited to approximately 32KB. The actual limit depends on the data type and the exact amount of header data required for each element. Be sure to use appropriate error checking (as demonstrated in the following examples) if your %List data is likely to approach this limit.

The examples in this section assume the following Caché class:

```
Class Sample.List Extends %Persistent
{
Property CurrentList As %List;

Method InitList() As %List {
   q $LB(1,"hello",3.14) }

Method TestList(NewList As %List) As %Integer {
   set $ZTRAP="ErrTestList"
   set ItemCount = $LISTLENGTH(NewList)
   if (ItemCount = 0) {set ItemCount = -1}
   q ItemCount
ErrTestList
   set $ZERROR = ""
   set $ZTRAP = ""
   q 0 }
}
```

The **TestList()** method is used to test if a Perl array is a valid Caché list. If the list is too large,the method traps an error and returns `0` (Perl false). If the list is valid, it returns the number of elements. If a valid list has `0` elements, it returns `-1`.

Example 1: Caché to Perl

The following code creates a Sample.List object, gets a predefined Caché list from the **InitList()** method, transforms it into a Perl array, and displays information about the array:

```
$listobj = $database->create_new("Sample.List",undef);
$arrayref = $listobj->InitList();
@array = @$arrayref;

print "Initial List from Cache:\n";
print "array address = $arrayref\n";
print "array contents = @array\n";
print "There are ",scalar(@array)," elements in the list:\n";
for ($i=0;$i<scalar(@array);$i++) {
   print " element ",$i+1," = [$array[$i]]\n";
}
```

This code produces the following output:

```
Initial List from Cache:
array address = ARRAY(0x18d3d04)
array contents = Cache to Perl: hello! 1 3.14
There are 4 elements in the list:
   element 1 = [Cache to Perl: hello!]
   element 2 = [1]
   element 3 = []
   element 4 = [3.14]
```

In element 3, the `null` list element in Caché corresponds to value of `undef` in the Perl array.

Example 2: Perl to Caché and Back Again

The following code passes a list in both directions. It creates a small Perl array, stores it in the Caché object's CurrentList property, then gets it back from the property and converts it back to a Perl array.

```
   @oldarray = (1, undef, 2.78,"Just a small list.");
   $listobj->set("CurrentList",\@oldarray);
   $arrayref = $listobj->get("CurrentList");
   @newarray = @$arrayref;

   print "\n\nThis list is from property CurrentList:\n";
   print "There are ",scalar(@newarray)," elements in the list:\n";
   for ($i=0;$i<scalar(@newarray);$i++) {
      print " element ",$i+1," = [$newarray[$i-1]]\n";
   }
   print "\nThe old and new arrays ";
   if ("@oldarray" eq "@newarray")
       {print "match:\n"}
   else
      {print "DON'T match:\n"};
   print "   old:>@oldarray<\n",
         "   new:>@newarray<\n";
```

This code produces the following output:

```
   This list is from property CurrentList:
   There are 4 elements in the list:
     element 1 = [Just a small list.]
     element 2 = [1]
     element 3 = []
     element 4 = [2.78]

   The old and new arrays match:
      old:>1  2.78 Just a small list.<
      new:>1  2.78 Just a small list.<
```

Example 3: Testing List Capacity

It is important to make sure that a Cache %List variable can contain the entire Perl array. The following code creates a Perl array that is too large, and attempts to store it in the CurrentList property.

```
# Create a large array and print array information.
   $longitem = "1022 character element".(1234567890 x 100);
   @array =("This array is too long.");
   $cache_list_size = length($array[0]);
   for ($i=0;$i<33;$i++) {
        push(@array,$longitem);
        $cache_list_size += length($longitem);
   };
   print "\n\nNow for a HUGE list:\n";
   print "Total bytes required by Cache' list: more than $cache_list_size\n";
   print "There are ",scalar(@array)," elements in the ingoing list.\n\n";

# Check to see if the array will fit.
   $bool =  $listobj->TestList(\@array);
   print "TestList() reports that the array ";
   if ($bool) {print "is OK, and has $bool elements.\n";}
   else {print "will be damaged by the conversion.\n"};

# Pass the array to Cache', get it back, and display the results
   $listobj->set("CurrentList",\@array);
   $badarrayref = $listobj->get("CurrentList");
   @badarray = @$badarrayref;

   print "There are ",scalar(@badarray)," elements in the returned list:\n";
   for ($i=0;$i<scalar(@badarray);$i++) {
      $line = $badarray[$i];
      # long elements are shortened for readability.
      if (length($line)> 1000) {substr($line,10,980) = "..."};
      print "   element ",$i+1," = [$line]\n";
   }
```

The printout shortens undamaged sections of the long elements to make the output more readable. The following output is produced on a unicode system:

```
   Now for a HUGE list:
   Total bytes required by Cache' list: more than 33749
   There are 34 elements in the ingoing list.

   TestList() reports that the array will be damaged by the conversion.
   There are 17 elements in the returned list:
      element 1 = [This array is too long.]
      element 2 = [1022 chara...90123456789012345678901234567890]
      element 3 = [1022 chara...90123456789012345678901234567890]
      element 4 = [1022 chara...90123456789012345678901234567890]
```

```
element 5 = [1022 chara...90123456789012345678901234567890]
element 6 = [1022 chara...90123456789012345678901234567890]
element 7 = [1022 chara...90123456789012345678901234567890]
element 8 = [1022 chara...90123456789012345678901234567890]
element 9 = [1022 chara...90123456789012345678901234567890]
element 10 = [1022 chara...90123456789012345678901234567890]
element 11 = [1022 chara...90123456789012345678901234567890]
element 12 = [1022 chara...90123456789012345678901234567890]
element 13 = [1022 chara...90123456789012345678901234567890]
element 14 = [1022 chara...90123456789012345678901234567890]
element 15 = [1022 chara...90123456789012345678901234567890]
element 16 = [1022 chara...90123456789012345678901234567890]
element 17 = [1022 chara...90123456ray is too long.ï´ È‡1022 c]
```

The damaged list contains only 17 of the original 34 elements, and element 17 is corrupted.

2.4 Using Relationships

Relationships are supported through the relationship object and its methods.

The following example generates a report by using the one/many relationship between the `$company` object and a set of `$employee` objects. The relationship object `$emp_relationship` allows the code to access all `$employee` objects associated with the `$company` object:

```
$company = $database->openid("Sample.Company", "1", -1, 0);
$emp_relationship = $company->get("Employees");
$index=undef;
do {
   $employee = $emp_relationship->run_obj_method("GetNext",$index);
   # "GetNext" sets $index to the next valid index, or to undef if
   # there are no more records.
   if ($index) {
      $name = $employee->get("Name");
      $title = $employee->get("Title");
      $company = $employee->get("Company");
      $compname = $company->get("Name");
      $SSN = $employee->get("SSN");
      print "index=$index employee name=$name SSN=$SSN ",
            "title=$title companyname=$compname\n";
   };
} while ($index);
```

The following code creates a new employee record, adds it to the relationship, and automatically saves the employee information when it saves `$company`.

```
$new_employee = $database->create_new("Sample.Employee","");
$new_employee->set("Name",$name);
$new_employee->set("Title",$title);
$new_employee->set("SSN",$SSN);
$emp_relationship->run_obj_method("Insert", $new_employee);
$company->run_obj_method("%Save");
```

2.5 Using Queries

The basic procedure for running a query is as follows:

- *Create the query object*

  ```
  $query = $database->alloc_query()
  ```

  where `$database` is an **Intersys::PERLBIND::Database** object and `$query` is an **Intersys::PERLBIND::Query** object.

- *Prepare the query*

An SQL query uses the **prepare()** method:

```
$query->prepare($sql_query, $sqlcode)
```

where `$sql_query` is a string containing the query to be executed.

A Caché class query uses the **prepare_class()** method:

```
$query->prepare_class($class_name, $query_name, $sqlcode)
```

- *Assign parameter values*

  ```
  $query->set_par($idx, $val)
  ```

  The **set_par()** method assigns value `$val` to parameter `$idx`. The value of an existing parameter can be changed by calling **set_par()** again with the new value. The Query class provides several methods that return information about an existing parameter.

- *Execute the query*

  ```
  $query->execute($sqlcode)
  ```

  The **execute()** method generates a result set using any parameters defined by calls to **set_par()**.

- *Fetch a row of data*

  ```
  @data_row = $query->fetch($sqlcode)
  ```

  Each call to the **fetch()** method retrieves a row of data from the result set and returns it as a list. When there is no more data to be fetched, it returns an empty list. The Query class provides several methods that return information about the columns in a row.

  Here is a simple SQL query that retrieves data from Sample.Person:

```
# Allocate, prepare, set a parameter, and get a result set:
$query = $database->alloc_query();
$sql_query =
   "SELECT ID, Name, DOB, SSN FROM Sample.Person
   ~WHERE Name %STARTSWITH ?";
$sqlcode=0;
$query->prepare($sql_query, $sqlcode);
$query->set_par(1,"A");
$query->execute($sqlcode);

# Fetch each row in the result set, and print the
# name and value of each column in a row:
$rownum = 1;
while (@data_row = $query->fetch($sqlcode)) {
   print "Row $rownum ==================\n";
   $colnum = 1;
   foreach $col_data (@data_row) {
      $col_name = $query->col_name($colnum);
      print " $col_name: $col_data\n";
      $colnum++;
   }
   $rownum++
}
```

For more information on the Intersys::PERLBIND::Query class, see Queries in the Perl Client Class Reference chapter. For information about queries in Caché, see Defining and Using Class Queries in *Using Caché Objects*.

2.6 Using %Binary Data

The Perl binding uses the Perl **pack()** and **unpack()** functions to convert data between a Caché %Binary and a Perl array of ords. Each byte of the Caché binary data is represented in Perl as an integer between 0 and 255.

The examples in this section assume the following Caché class:

```
Class Sample.Bin Extends %Persistent
{
   Property B As %Binary;

   Method GetBinary() As %Binary {
      q "hello"}

   Method SetBinary(PerlBin As %Binary) {
      s ^foobar=PerlBin}
}
```

The class uses %Binary data in a variety of ways: as a property value, a method return value, a method argument, and a global variable. The following code transfers binary data back and forth between Caché and Perl several times. The Perl application makes a change in the data each time.

The first input is a method result. The **GetBinary()** method returns the %Binary value `"hello"`. The value is unpacked and the resulting array is printed out:

```
$binobj = $database->openid("Sample.Bin",1,-1,0);
$B_packed = $binobj->run_obj_method("GetBinary");
@B_unpacked = unpack("c*",$B_packed ); # "c*" is the template for an ord
foreach $c (@B_unpacked) {
   # "%c" turns the ord value $c into a character
   printf "[%c]",$c
}
print "\n";
```

The first line of output looks like this:

```
[h][e][l][l][o]
```

Now the program changes the first byte to a `"j"` and repacks the array. The packed value is stored in Caché %Binary property `B`, and the altered Caché object is saved:

```
$B_unpacked[0] = ord("j");
$B_packed = pack("c*",@B_unpacked);
$binobj->set("B",$B_packed);
$binobj->run_obj_method("%Save");
```

The second input is a property value. The %Binary value `"jello"` is retrieved from property `B` and unpacked. The program changes the value again, this time by adding a new element to the end of the array:

```
$B_packed = $binobj->get("B");
@B_unpacked = unpack("c*",$B_packed ); # "c*" is the template for an ord
push @B_unpacked, ord("!");
foreach $c (@B_unpacked) {
   printf "[%c]",$c
}
print "\n";
};
```

Combined with the previous line, the output now look like this:

```
[h][e][l][l][o]
[j][e][l][l][o][!]
```

Finally, the program packs the array again and passes it as an argument to the **SetBinary()** method, which stores it in Caché global variable `^foobar`:

```
$B_packed = pack("c*",@B_unpacked);
$binobj->run_obj_method("SetBinary",$B_packed);
```

2.7 Handling Exceptions

The Perl binding uses Perl exceptions to return errors from the C binding and elsewhere.

Here are some examples of using Perl exceptions:

```
eval {
   $answer = $variant2->run_obj_method("PassLastByRefAdd17","10","goodbye");
};
if ($@) {
   print "Perl exception $@\n";
}

# Attempt to open a nonexistent id
eval {
   $causeException = $database2->openid("NonExistent", "1", -1, 0);
};
if ($@) {
   print "Perl exception $@\n";
}
```

2.7.1 Handling %Status Return Values

When a method declares a return type of %Status, the status value is not returned. Instead, an exception is thrown. For example, assume that we want to call the following Caché method that returns %Status:

```
Class User.PerlStatus Extends %Persistent{
   ClassMethod MyStatus() as %Status
   {
      q $$$ERROR(-137,"bad fine structure constant")
   }
}
```

The following Perl binding program calls the **MyStatus()** method, captures the exception, and prints the return status value:

```
# TestStatus.pl: illustrate handling a bad status return from COS

use Intersys::PERLBIND;
my $user="_SYSTEM";
my $password="SYS";
my $url = "localhost[1972]:USER";
eval {
   $conn = Intersys::PERLBIND::Connection->new($url,$user,$password,0);
   $database = Intersys::PERLBIND::Database->new($conn);
   $status=$database->run_class_method("User.PerlStatus", "MyStatus");
   # exception is thrown before we get here
   print "status=$status\n";
};

if ($@) {
   print "Caught exception: $@\n";
} else {
   print "\nSample finished running\n";
}
```

2.7.2 Error Reporting

When processing an argument or a return value, error messages from the C binding are specially formatted by the Perl binding layer. This information can be used by InterSystems WRC to help diagnose the problem.

Here is a sample error message:

```
file=PERLBIND.xs line=71 err=-1 message=cbind_variant_set_buf()
cpp_type=4 var.cpp_type=-1 var.obj.oref=1784835886
class_name=%Library.RelationshipObject mtd_name=GetNext argnum=0
```

The error message components are:

message	meaning
`file=PERLBIND.xs`	file where the failure occurred.
`line=71`	line number in the file.
`err=-1`	return code from the C binding.
`message= cbind_variant_set_buf()`	C binding error message.
`cpp_type=4`	cpp type of the method argument or return type.
`var.cpp_type=-1`	variant cpp type.
`var.obj.oref=1784835886`	variant oref.
`class_name= %Library.RelationshipObject`	class name of the object on which the method is invoked.
`mtd_name=GetNext`	method name.
`argnum=0`	argument number. 0 is the first argument and -1 indicates a return value.

2.8 Perl Binding Constraints and Limitations

The Perl binding imposes the following limitations, which are usually related to differences in the way that Perl and Caché handle certain datatypes:

- The Perl binding inherits the limitations of ObjectScript methods and datatypes. For example, if your Caché system limits %String to 32K, a Perl program would fail if it attempted to write more than 32K to a Caché character stream.
- Caché and Perl differ on the numeric ranges of fundamental data types and on parameters such as the range of the mantissa and the exponent. This may cause problems when your program manipulates very large numbers (for example, comparing integers with greater than about 15 digits). A workaround is to fetch such data from Caché as %Binary strings.
- When working with both 32-bit and 64-bit systems, it is important to remember that numeric processing may be affected by 32-bit limitations. On a 32-bit system, an integer variable is limited to 32 bits, and a decimal must fit both signficand and exponent into 32 bits as signed integers.
- A %Status parameter cannot be passed by reference (also see Handling %Status Return Values). A workaround is to pass the value as an integer.
- Some Caché library methods receive and return Caché local arrays as parameters passed by reference. Caché language bindings do not support methods that have local array parameters. A workaround is to write Caché wrapper methods that process the local arrays on the Caché side.
- A connection will be lost if the Perl code forks to a child process (or Windows thread). For example, the following code fragment forks a no-op child process, after which the connection is lost by the main program. The connection reference is copied for the child process and closed when the child exits.

```
my $pid = fork();
if ( !defined( $pid)) {
   die( "fork() failed\n");
}
if ( $pid == 0) { # child
   print( "child: calling exit()\n");
   exit( 0);
} else { # parent
   waitpid( $pid, 0); # wait for the child to exit
}
my $ret = $db->run_class_method( "%SYSTEM.OBJ", "Version", ());
```

3

Perl Client Class Reference

This chapter describes how Caché classes and datatypes are mapped to Perl code, and provides details on the classes and methods supported by the Caché Perl binding. The following subjects are discussed:

- Datatypes — %Binary data and Null support.
- Connections — methods used to create a physical connection to a namespace in a Caché database.
- Database — methods used to open or create Caché objects, create queries, and run Caché class methods.
- Objects — methods used to manipulate Caché objects by getting or setting properties, running object methods, and returning information about the objects.
- Queries — methods used to run queries and fetch the results.
- Times and Dates — methods used to access the Caché %Time, %Date, or %Timestamp datatypes.
- Locale and Client Version — methods that provide access to Caché version information and Windows locale settings.

3.1 Datatypes

All Caché datatypes are supported. See the following sections for information on specific datatypes:

- The Caché %Binary datatype corresponds to a Perl array of ords. See Using %Binary Data for examples.
- Collections such as %Library.ArrayOfDataTypes and %Library.ListOfDataTypes are handled through object methods of the Caché %Collection classes. See %Collection Objects for examples.
- Caché %Library.List variables are mapped to Perl array references. See %List Variables for examples.
- Caché %Time, %Date, and %Timestamp datatypes are supported by corresponding classes in the Perl binding. See Times and Dates for a description of these classes.

Limitations

In some cases, the Caché language binding engine will map a Perl variable to an intermediate datatype that does not precisely match a Caché datatype. This increases processing efficiency, but can impose minor constraints on the way the datatype is used. See Perl Binding Constraints and Limitations for details.

Null support

In the Perl binding, Perl `undef` corresponds to Caché `null` (i.e., ""). Perl has the concept of an empty string, but the Perl binding will not interpret this as a Caché null.

3.2 Connections

Methods of the Intersys::PERLBIND::Connection package create a physical connection to a namespace in a Caché database. A Connection object is used only to create a Database object, which is the logical connection that allows Perl binding applications to manipulate Caché objects. See Connecting to the Caché Database for information on how to use the Connection methods.

Here is a complete listing of Connection methods:

new()

```
$conn = Intersys::PERLBIND::Connection->new($url, $user, $pwd, $timeout)
```

See Connection Information later in this section for a detailed discussion of the parameters.

new_secure()

```
$conn = Intersys::PERLBIND::Connection->new_secure($url, $srv_principal,
    $security_level, $timeout)
```

Connection->new_secure() returns the connection proxy that is used to get the proxy for the Caché namespace identified by `$url`. This method takes the following parameters:

- `$url` — See Connection Information later in this section for a detailed description of the URL format.
- `$srv_principal` — A Kerberos "principal" is an identity that is represented in the Kerberos database, has a permanent secret key that is shared only with the Kerberos KDCs (key distribution centers), can be assigned credentials, and can participate in the Kerberos authentication protocol.
  - A "user principal" is associated with a person, and is used to authenticate to services which can then authorize the use of resources (for example, computer accounts or Caché services).
  - A "service principal" is associated with a service, and is used to authenticate user principals and can optionally authenticate itself to user principals.
  - A "service principal name" (such as `$srv_principal_name`) is the string representation of the name of a service principal, conventionally of the form:

    <service>/<instance>@<REALM>

    For example:

    cache/turbo.iscinternal.com@ISCINTERNAL.COM

  On Windows, The KDCs are embedded in the domain controllers, and service principal names are associated with domain accounts.

  See your system's Kerberos documentation for a detailed discussion of principals.
- `$security_level` — Sets the "Connection security level", which is an integer that indicates the client/server network security services that are requested or required. Security level can take the following values:
  - `0` — None.
  - `1` — Kerberos client/server mutual authentication, no protection for data.
  - `2` — As `1`, plus data source and content integrity protection.
  - `3` — As `2`, plus data encryption.
- `$timeout` — Number of seconds to wait before timing out.

3.2.1 Connection Information

The following information is needed to establish a connection to the Caché database:

- *URL* — The URL specifies the server and namespace to be accessed as a string with the following format:

  ```
  <address>[<port>]:<namespace>
  ```

  For example, the sample programs use the following connection string:

  ```
  "localhost[1972]:Samples"
  ```

  The components of this string are:

  – `<address>` — The server TCP/IP address or Fully Qualified Domain Name (FQDN). The sample programs use `"localhost"` (`127.0.0.1`), assuming that both the server and the Perl application are on the same machine.

  – `<port>` — The server TCP/IP port number for this connection. Together, the IP address and the port specify a unique Caché server.

  – `<namespace>` — The Caché namespace containing the objects to be used. This namespace must have the Caché system classes compiled, and must contain the objects you want to manipulate. The sample programs use objects from the SAMPLE namespace.

- *username* — The username under which the connection is being made. The sample programs use `"_SYSTEM"`, the default SQL System Manager username.
- *password* — The password associated with the specified username. Sample programs use the default, `"SYS"`.

3.3 Database

Database objects provide a logical connection to a Caché namespace. Methods of the Intersys::PERLBIND::Database package are used to open or create Caché objects, create queries, and run Caché class methods. Database objects are created by the **Database->new()** method. See Connecting to the Caché Database for more information on creating a Database object.

Here is a complete listing of Database methods:

alloc_query()

```
$query = $database->alloc_query()
```

Create a new Query object. See Queries for details on Query object methods.

create_new()

```
$obj = $database->create_new($type, $init_val)
```

Creates a new Caché object instance from the class named by `$type`. Normally, `$init_val` is `""`. See Objects for details on the objects created with this method.

new()

```
$database = Intersys::PERLBIND::Database->new($conn)
```

Creates a Database object, which provides a logical connection to a Caché namespace. The Connection object `$conn` supplies a physical connection to the namespace that the Database object will access.

open()

```
$obj = $database->open($class_name, $oid, $concurrency, $timeout, $res)
```

Opens a Caché object instance using the class named by `$class_name` and the oid of the object. The `$concurrency` argument has a default value of `-1`. `$timeout` is the ODBC query timeout.

openid()

```
$obj = $database->openid($class_name, $id, $concurrency, $timeout)
```

Opens a Caché object instance using the class named by `$class_name` and the id of the object. The `$concurrency` argument has a default value of `-1`. `$timeout` is the ODBC query timeout.

run_class_method()

```
$value = $database->run_class_method($class_name, $method_name, LIST)
```

Runs the class method `$method_name`, which is a member of the `$class_name` class in the namespace that `$database` is connected to. Arguments are passed in `LIST`. Some of these arguments may be passed by reference depending on the class definition in Caché. Return values correspond to the return values from the Caché method.

3.4 Objects

Methods of the Intersys::PERLBIND::Object package provide access to a Caché object. An Object object is created by the Intersys::PERLBIND::Database **create_new()** method (see Database for a detailed description). See Using Caché Object Methods for information on how to use the Object methods.

Here is a complete listing of Object methods:

get()

```
$value = $object->get($prop_name)
```

Returns the value of property `$prop_name` in Caché object `$object`.

get_methods()

```
@methodlist = $object->get_methods();
$methodcount = $object->get_methods();
```

In list context, returns a list containing the names of methods in Caché object `$object`. In scalar context, returns the number of methods in `$object`.

get_properties()

```
@proplist = $object->get_properties();
$propcount = $object->get_properties();
```

In list context, returns a list containing the names of properties in Caché object `$object`. In scalar context, returns the number of properties in `$object`. Private and multidimensional properties are not returned, and are not accessible through the Perl binding.

is_method()

```
$bool = $object->is_method($name);
```

Returns `1` if `$name` is a valid method name for `$object`, or `0` otherwise.

is_property()

```
$bool = $object->is_property($name);
```

Returns `1` if `$name` is a valid property name for `$object`, or `0` otherwise.

run_obj_method()

```
$value = $object->run_obj_method($method_name, LIST)
```

Runs method `$method_name` on Caché object `$object`. Arguments are passed in `LIST`. Some of these arguments may be passed by reference depending on the class definition in Caché. Return values correspond to the return values from the Caché method.

set()

```
$object->set($prop_name, $val)
```

Sets property `$prop_name` in Caché object `$object` to `$val`.

3.5 Queries

Methods of the Intersys::PERLBIND::Query package provide the ability to prepare a query, set parameters, execute the query, and and fetch the results. A Query object is created by the Intersys::PERLBIND::Database **alloc_query()** method (see Database for a detailed description). See Using Queries for information on how to use the Query methods.

Here is a complete listing of Query methods:

3.5.1 prepare query

prepare()

```
$query->prepare($string, $sqlcode)
```

Prepares a query using the SQL string in `$string`.

prepare_class()

```
$query->prepare_class($class_name, $query_name, $sqlcode)
```

Prepares a query in a class definition

3.5.2 set parameters

set_par()

```
$query->set_par($idx, $val)
```

Assigns value `$val` to parameter `$idx`. The method can be called several times for the same parameter. The previous parameter value will be lost, and the new value can be of a different type. The **set_par()** method does not support by-reference parameters.

is_par_nullable()

```
$nullable = $query->is_par_nullable($idx)
```

Returns `1` if parameter `$idx` is nullable, else `0`.

is_par_unbound()

```
$unbound = $query->is_par_unbound($idx)
```

Returns `1` if parameter `$idx` is unbound, else `0`.

num_pars()

```
$num = $query->num_pars()
```

Returns number of parameters in query.

par_col_size()

```
$size = $query->par_col_size($idx)
```

Returns size of parameter column.

par_num_dec_digits()

```
$num = $query->par_num_dec_digits($idx)
```

Returns number of decimal digits in parameter.

par_sql_type()

```
$type = $query->par_sql_type($idx)
```

Returns sql type of parameter.

3.5.3 execute query

execute()

```
$query->execute($sqlcode)
```

Generates a result set using any parameters defined by calls to **set_par()**.

3.5.4 fetch results

fetch()

```
@data_row = $query->fetch($sqlcode)
$column_count = scalar($query->fetch($sqlcode))
```

When run in a list context, the method retrieves a row of data from the result set and returns it as a list. When there is no more data to be fetched, it returns an empty list.

When run in a scalar context, the method returns the number of columns in a row.

col_name()

```
$name = $query->col_name($idx)
```

Returns name of column.

col_name_length()

```
$length = $query->col_name_length($idx)
```

Returns length of column name.

col_sql_type()

```
$sql_type = $query->col_sql_type($idx)
```

Returns sql type of column.

num_cols()

```
$num_cols = $query->num_cols()
```

Returns number of columns in query.

3.6 Times and Dates

The PTIME_STRUCTPtr, PDATE_STRUCTPtr, and PTIMESTAMP_STRUCTPtr packages are used to manipulate Caché %TIME, %DATE, or %TIMESTAMP datatypes.

3.6.1 %TIME

Methods of the PTIME_STRUCTPtr package are used to manipulate the Caché %DATE data structure. Times are in `hh:mm:ss` format. For example, 5 minutes and 30 seconds after midnight would be formatted as `00:05:30`. Here is a complete listing of Time methods:

new()

```
$time = PTIME_STRUCTPtr->new()
```

Create a new Time object.

get_hour()

```
$hour = $time->get_hour()
```

Return hour

get_minute()

```
$minute = $time->get_minute()
```

Return minute

get_second()

```
$second = $time->get_second()
```

Return second

set_hour()

```
$time->set_hour($hour)
```

Set hour (an integer between 0 and 23, where 0 is midnight).

set_minute()

```
$time->set_minute($minute)
```

Set minute (an integer between 0 and 59).

set_second()

```
$time->set_second($second)
```

Set second (an integer between 0 and 59).

toString()

```
$stringrep = $time->toString()
```

Convert the time to a string: `hh:mm:ss`.

3.6.2 %DATE

Methods of the PDATE_STRUCTPtr package are used to manipulate the Caché %DATE data structure. Dates are in `yyyy-mm-dd` format. For example, December 24, 2003 would be formatted as `2003-12-24`. Here is a complete listing of Date methods:

new()

```
$date = PDATE_STRUCTPtr->new()
```

Create a new Date object.

get_year()

```
$year = $date->get_year()
```

Return year

get_month()

```
$month = $date->get_month()
```

Return month

get_day()

```
$day = $date->get_day()
```

Return day

set_year()

```
$date->set_year($year)
```

Set year (a four-digit integer).

set_month()

```
$date->set_month($month)
```

Set month (an integer between 1 and 12).

set_day()

```
$date->set_day($day)
```

Set day (an integer between 1 and the highest valid day of the month).

toString()

```
$stringrep = $date->toString()
```

Convert the date to a string: `yyyy-mm-dd`.

3.6.3 %TIMESTAMP

Methods of the PTIMESTAMP_STRUCTPtr package are used to manipulate the Caché %TIMESTAMP data structure. Timestamps are in `yyyy-mm-dd`<space>`hh:mm:ss.fffffffff`. format. For example, December 24, 2003, five minutes and 12.5 seconds after midnight, would be formatted as:

```
2003-12-24 00:05:12:500000000
```

Here is a complete listing of TimeStamp methods:

new()

```
$timestamp = PTIMESTAMP_STRUCTPtr->new()
```

Create a new Timestamp object.

get_year()

```
$year = $timestamp->get_year()
```

Return year in `yyyy` format.

get_month()

```
$month = $timestamp->get_month()
```

Return month in `mm` format.

get_day()

```
$day = $timestamp->get_day()
```

Return day in `dd` format.

get_hour()

```
$hour = $timestamp->get_hour()
```

Return hour in `hh` format.

get_minute()

```
$minute = $timestamp->get_minute()
```

Return minute in `mm` format.

get_second()

```
$second = $timestamp->get_second()
```

Return second in `ss` format.

get_fraction()

```
$fraction = $timestamp->get_fraction()
```

Return fraction of a second in `fffffffff` format.

set_year()

```
$timestamp->set_year($year)
```

Set year (a four-digit integer).

set_month()

```
$timestamp->set_month($month)
```

Set month (an integer between 1 and 12).

set_day()

```
$timestamp->set_day($day)
```

Set day (an integer between 1 and the highest valid day of the month).

set_hour()

```
$timestamp->set_hour($hour)
```

Set hour (an integer between 0 and 23, where 0 is midnight).

set_minute()

```
$timestamp->set_minute($minute)
```

Set minute (an integer between 0 and 59).

set_second()

```
$timestamp->set_second($second)
```

Set second (an integer between 0 and 59).

set_fraction()

```
$timestamp->set_fraction($fraction)
```

Set fraction of a second (an integer of up to nine digits).

toString()

```
$stringrep = $timestamp->toString()
```

Convert the timestamp to a string `yyyy-mm-dd hh:mm:ss.fffffffff`.

3.7 Locale and Client Version

Methods of the Intersys::PERLBIND default package provide access to Caché version information and Windows locale settings. Here is a complete listing of these methods:

get_client_version()

```
$clientver = Intersys::PERLBIND::get_client_version();
```

Identifies the version of Caché running on the Perl client machine.

setlocale()

```
$newlocale = Intersys::PERLBIND::setlocale($category, $locale)
```

Sets the default locale and returns a locale string for the new locale. For example:

```
$newlocale = Intersys::PERLBIND::setlocale(0, "Russian") # 0 stands for LC_ALL
```

would set all locale categories to Russian and return the following string:

```
Russian_Russia.1251
```

If the `$locale` argument is an empty string, the current default locale string will be returned. For example, given the following code:

```
Intersys::PERLBIND::setlocale(0, "English")
$mylocale = Intersys::PERLBIND::setlocale(0, ""),"\n";
```

the value of `$mylocale` would be:

```
English_United States.1252
```

For detailed information, including a list of valid `$category` values, see the MSDN library (http://msdn.microsoft.com/library) entry for the **setlocale()** function in the Visual C++ runtime library.

set_thread_locale()

```
Intersys::PERLBIND::set_thread_locale($lcid)
```

Sets the locale id (LCID) for the calling thread. Applications that need to work with locales at runtime should call this method to ensure proper conversions. It is equivalent to calling **setlocale()** from the C++ runtime library inside the current thread, but a direct call to **setlocale()** may fail without throwing an exception in some cases.

For a listing of valid LCID values, see the "Locale ID (LCID) Chart" in the MSDN library (http://msdn.microsoft.com/library). The chart can be located by a search on "LCID Chart". It is currently located at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vsmsclcid.asp

For detailed information on locale settings, see the MSDN library entry for the **SetThreadLocale()** function, listed under "National Language Support".