Learn the OpenAccess API
Using Python

©Silicon Integration Initiative
Initial Contribution By

James Masters
Intel - 2013
Updates & Additions
by Silicon Integration Initiative - 2020
Lab 1.1

- This should have been completed in the Training Set Up Module
- The lab simply looks for the libraries for the Python API to insure the installation is correct

see: labs/1.1/test.py
oaScript Overview

• The only access to OA data is through the C++ API
  – Write your own C++ application (longer formal development)
  – Or... oaScript is used to provide direct access through perl, python, ruby, or tcl (rapid prototyping and application development)
  – Interface to C++ API is handled using SWIG (www.swig.org)
Common Themes in Wrapped Languages

• Common structure/intent
  – Remain as close to the OA API as possible to allow reuse of the existing C++ OA documentation
  – Wrapped names may be slightly adjusted to meet language requirements (as needed)
  – Use type conversions where it makes sense

• Unique behaviors
  – Type conversion customizations for ease-of-use and match language data types where it makes sense
  – Matching idioms of a language
Section 2 Introduction

- The OpenAccess API has over 2000 classes
- The average user uses a small subset which we will cover in this course
- It is a good idea to know how to look up new things within the Exhaustive C++ API documentation
Class Listing

There are approximately 2000 OA classes... don’t be overwhelmed! You will probably end up only using less than a hundred.
Lab 2.1: Using OA API Doc

• Use the API to determine how to create a polygon (oaPolygon)
  – How do you get the number of points in the polygon after it’s created?
  – What is the return value for isOrthogonal()?
  – What is the return value for setPoints()?
The API Documentation

- Lists all functions (including inherited)
- Inheritance tree (clickable)
- Instance Methods (click for details)
- Static/Class Methods (click for details)
- Detailed Function Description further below
Mapping to OA API Documentation

- Only slight changes made to API where necessary in target languages
- Navigate to the “oaLib” class “create” method in the API document

**Static Public Methods**

```cpp
oaLib * create (const oaScalarName &name, const oaString &libPath, oaLibMode mode=oacSharedLibMode, const oaString &dmsystem="oaDMSystem", const oaDMDAtrArray *dmAttrList=NULL)
```

<table>
<thead>
<tr>
<th>Language</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perl</td>
<td>new oa::oaLib::create(oaScalarName, string, oaLibMode, string, array</td>
</tr>
<tr>
<td>Python</td>
<td>oa.oaLib.create(oaScalarName, oaString, oaLibMode, oaString, array</td>
</tr>
<tr>
<td>Ruby</td>
<td>Oa::OaLib.create(oaScalarName</td>
</tr>
<tr>
<td>Tcl</td>
<td>oa::oaLib_create oaScalarName string</td>
</tr>
</tbody>
</table>
Mapping to OA API Documentation

- Navigate to the “oaLib” class “getAccess” method in the API document

**Public Methods**

`oaBoolean getAccess (oaLibAccess accessType, oaUInt4 ,meout=0)`

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
</table>
| Perl     | ```
/lib->getAccess(new oa::oaLibAccess('read'));
/lib->getAccess($oa::oacReadLibAccess);
``` |
| Python   | ```
lib.getAccess(oa.oaLibAccess('read'))
lib.getAccess(oa.oaLibAccess(oa.oacReadLibAccess))
``` |
| Ruby     | ```
lib.getAccess(Oa::OaLibAccess.new('read'))
lib.getAccess(Oa::OaLibAccess.new(Oa::OacReadLibAccess))
lib.getAccess(:read)
``` |
| Tcl      | ```
/lib getAccess [oa::oaLibAccess "read"]
/lib getAccess [oa::oaLibAccess $oa::oacReadLibAccess]
/lib getAccess $oa::oacReadLibAccess
``` |
Lab 2.2

Goal - Learn to use the documentation and demonstrate ability to navigate the C++ API and translate into the native language format.
Write a script to:
1. Create an oaTimer object
2. Create some kind of delay
3. Get the elapsed number of seconds for the operation
4. print the value

Compare your script to labs/2.2/timer.py
## Basic Type Mapping
(conversion from a basic OA type to a native language type)

<table>
<thead>
<tr>
<th>OA Type</th>
<th>Perl</th>
<th>Python</th>
<th>Ruby</th>
<th>TCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>void/NULL</td>
<td>undef</td>
<td>None</td>
<td>nil</td>
<td>“NULL”</td>
</tr>
<tr>
<td>oaBoolean</td>
<td>Integer (scalar)</td>
<td>bool</td>
<td>TrueClass/FalseClass</td>
<td>Integer</td>
</tr>
<tr>
<td>oa*Int</td>
<td>Integer (scalar)</td>
<td>int</td>
<td>Fixnum</td>
<td>Integer</td>
</tr>
<tr>
<td>oaFloat/Double</td>
<td>Float (scalar)</td>
<td>float</td>
<td>Float</td>
<td>Float</td>
</tr>
<tr>
<td>oaString</td>
<td>String (scalar)</td>
<td>oaString or str</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>oa*Array</td>
<td>oa*Array or native array</td>
<td>oa*Array or native array</td>
<td>Oa*Array or native array</td>
<td>oa*Array or native array</td>
</tr>
<tr>
<td>oaTime</td>
<td>oaTime</td>
<td>oaTime</td>
<td>Time</td>
<td>oaTime</td>
</tr>
<tr>
<td>oaTimestamp</td>
<td>oaTimeStamp</td>
<td>Int</td>
<td>Integer</td>
<td>oaTimeStamp</td>
</tr>
<tr>
<td>oaComplex</td>
<td>oaComplex</td>
<td>complex</td>
<td>OaComplex</td>
<td>oaComplex</td>
</tr>
<tr>
<td>oaPoint</td>
<td>oaPoint or (x,y) array</td>
<td>oaPoint or [x,y] array</td>
<td>OaPoint or [x,y] array</td>
<td>oaPoint or [x,y] array</td>
</tr>
<tr>
<td>oaBox</td>
<td>oaBox or [l,b,r,t] array</td>
<td>oaBox or [l,b,r,t] array</td>
<td>OaBox or [l,b,r,t] array</td>
<td>oaBox or [l,b,r,t] array</td>
</tr>
<tr>
<td>oaTransform</td>
<td>oaTransform or [x,y,o] array</td>
<td>oaTransform or [x,y,o] array</td>
<td>OaTransform or [x,y,o] array</td>
<td>oaTransform or [x,y,o] array</td>
</tr>
</tbody>
</table>
Enumeration Wrappers

- Enumerated wrappers represent a numerical value but can be retrieved using a string
- Navigate to “oaViewType” in the API

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
</table>
| Perl     | `oa::oaViewType::get('maskLayout')`  
           | `oa::oaViewType::get($oa::oacMaskLayout)` |
| Python   | `oa.oaViewType.get('maskLayout')`  
           | `oa.oaViewType.get(oa.oacMaskLayout)` |
| Ruby     | `Oa::OaViewType.get(:maskLayout)`  
           | `Oa::OaViewType.get('maskLayout')`  
           | `Oa::OaViewType.get(Oa::OacMaskLayout)` |
| Tcl      | `oa::oaViewType_get $oa::oacMaskLayout`  
           | `oa::oaViewType_get maskLayout` |
Output Arguments

• Some OA functions expect the return value to be passed through the argument list (argument is a pointer)
  – Function return value is C++ “void” (no return value)
  – You need to pre-allocate an empty object to be filled
  – Example: `void oaName::get(oaString &out)`

• As a convenience, oaScript does two things with these functions:
  1. The output argument also becomes the return value instead of not returning anything
  2. You do not need to pre-allocate the output argument

• Examples - both are equivalent in oaScript:
  – Option 1: as documented in the C++ API doc
    ```cpp
    name = oa.oaName("foo")
    str = oa.oaString()
    name.get(str) #=> returns "foo" and str="foo"
    ```
  – Option 2: without pre-allocation using oaScript:
    ```cpp
    name = oa.oaName("foo")
    name.get() #=> returns "foo"
    ```

Note: Python is the only language which defines `oaString()` since the native Python string is immutable (cannot be changed in a function). Pre-allocate other strings in the other languages using just a regular string.
Lab 2.3
• Goal - Get more familiar with using enumeration wrappers and also not needing to pre-allocate a string on output arguments.

• Create a script to:
  1. Create an oaViewType object from the "schematic" oaReservedViewType enumeration wrapper
  2. Print the view type object's name to the screen
     • From the object, not from the hard-coded "schematic" string

• Compare your script to labs/2.3/enum_view.py
What are the enumerated values for oaViewType?
  Hint: Look at oaReservedViewType
Section 2 Summary

- Using the C++ Documentation
- Type Mapping
- Output arguments
Silicon Integration Initiative

www.si2.org

For details contact Marshall Tiner
Director of Production Standards
mtiner@si2.org