Learn the OpenAccess API
Using Python

©Silicon Integration Initiative
Section 3 OA Basic Classes

- OpenAccess models just about everything as an object; including basic concepts like: point, box, array of points, etc.
- It is important to understand these basic concepts before working with design data
- All coordinate values are specified in Database Units (DBU)
  - Integer only values (no floating-point allowed)
  - DBU is what is stored on disk; conversion to user units is done by the application (you)
Note: For now, we will use integer values only for coordinates. This will be explained later in more detail.
• Points are represented in the `oaPoint` class:
  ```ruby
  point1 = oa.oaPoint(3, 4)
  point1.x() #=> 3
  point1.y() #=> 4
  point2 = oa.oaPoint(5, 6)
  ``
  ```ruby
  math.sqrt(point1.distanceFrom2(point2))
  #=> 2.8284271247461903
  ```

• As a convenience in `oaScript`, any function that accepts an `oaPoint` as an argument will also accept an array of exactly two integers which represent a x/y point:
  ```ruby
  math.sqrt(point1.distanceFrom2([9, 2]))
  #=> 6.324555320336759
  ```
oaBox

• Boxes are represented in the oaBox class:
  ```python
doctests
box = oa.oaBox(0, 0, 10, 10)
    box.lowerLeft() #=> oa.oaPoint(0, 0)
    box.upperRight() #=> oa.oaPoint(10, 10)
    box.getCenter() #=> oa.oaPoint(5, 5)
```

• As a convenience in oaScript, any function that accepts an oaBox as an argument will also accept an array of exactly four integers (left, bottom, right, top):
  ```python
box.contains([2, 2, 4, 4]) #=> True
```
Lab 3.1

- Goal - Become familiar with coordinates, points, and boxes
- Create a script to:
  1. Create two boxes in different locations
  2. Scale one of the boxes by 2X
  3. See if the boxes are touching each other
  4. See if a point is completely inside one of the boxes
     - Not outside or on an edge

- compare your script to labs/3.1/boxes.py
• Arrays of points are represented in the oaPointArray class. This is used to represent either a **closed set** of points (polygonal object) or an **open set** of points (line segments)

```ruby
poly = oa.oaPointArray([[0,0], [0,10], [5,10], [5,5], [10,5], [10,0]])
poly.getNumElements() #=> 6
poly.getArea() #=> 75.0
```

• As a convenience in oaScript, any function that accepts an `oaPointArray` as an argument will also accept an array of two element arrays of integers (points)
  – This is shown in the example above for the `oa.oaPointArray()` constructor where an existing set of points (array of arrays) is an `oaPointArray` to be copied and created into a new object
oaPointArray in other languages

- **Perl**
  ```perl
  $ptarr = new oa::oaPointArray( [[1,1], [1,2]] );
  foreach $point (@$ptarr) {
    printf("(%d, %d)\n", $$point[0], $$point[1]);
  }
  ```
- **Python**
  ```python
  ptarr = oa.oaPointArray(((1,1), (1,2)))
  for point in ptarr:
    print "(%d, %d)" % (point.x(), point.y())
  ```
- **Ruby**
  ```ruby
  ptarr = Oa::OaPointArray.new([[1,1], [1,2]])
  ptarr.each do |point|
    printf("(%d, %d)\n", point.x, point.y)
  end
  ```
- **Tcl**
  ```tcl
  oa::foreach point $ptarr {
    puts [format "(%d, %d)" [$point x] [$point y]]
  }
  ```
• In OA, objects can be transformed using the `oaTransform` class
  – Shifted by an x and y offset
  – Rotated about a reference point
  – Mirrored about the x or y axis
• See “Using Transforms” section of the API doc for details (Programmers Guide->Physical Design->Using Transforms)
• Example:

```python
point = oa.oaPoint(1,1)
xform = oa.oaTransform(oa.oacR90)
point.transform(xform)  #=> oaPoint(-1,1)
xform2 = oa.oaTransform(oa.oacR90, 2, 2)
point.transform(xform2)  #=> oaPoint(2,1)
```
Lab 3.2

• Goal - Become familiar with polygons and transforms
• Create a script to:
  • Create a polygon
  • Transform it in some way (rotate, mirror, and/or shift)
  • Print the resulting points to the screen
• compare your script to labs/3.2/points.py
oa*Name

- The oa*Name class provides a way to **store and translate** between different naming conventions
- Namespaces are used to manage handling of special characters or restrictions in naming
- The oaName concept is used for names like on nets and instances
- See “Name Mapping” section of the API doc for details
  - Programmers Guide->Names->Name Mapping

Name classes:
- **oaName** – wrapper to represent any of the oa*Name types
- **oaScalarName** – represents a non-bit name (e.g. “foo”)
- **oaVectorName** – represents a name with many bits (e.g. “foo[0:7]”)
- **oaVectorBitName** – represents a name with a single bit (e.g. “foo[1]”)
- **oaBundleName** – represents potentially repeated simple names
- **oaSimpleName** – wrapper to represent scalar, vector, or vector bit
The oaScalarName class is for single non-bit names
  - The most used out of all of the oa*Name classes
Example conversion between CDBA->Verilog
  - To highlight conversion, the special hierarchy character is used
    - Grave symbol for CDBA `
    - Period for Verilog .

cdba_ns = oa.oaCdbaNS()

v_ns = oa.oaVerilogNS()

name = oa.oaScalarName(cdba_ns, "a`b")

name.get(v_ns) #=> "a.b"
oaVectorName

- The oaVectorName class is for a range of bits
  - Bus bits separation character(s) are abstracted until a name is needed in a given namespace
- Example:

```java
vname = oa.oaVectorName("bit", 0, 7)  vname.getBaseName() #=> "bit"
vname.getStart() #=> 0
vname.getStop() #=> 7
vname.getStep() #=> 1
vname.getNumBits() #=> 8
   vname.get(oa.oaCdbaNS())#=>"bit<0:7>"
vname.get(oa.oaVerilogNS()) #=> "bit[0:7]"
```
The `oaVectorBitName` class is for a single bit
- Bus bits separation character(s) are abstracted until a name is needed in a given namespace

**Example:**
```ruby
bname = oa.oaVectorBitName("bit", 0)
  bname.getBaseName() #=> "bit"
bname.getIndex() #=> 0
bname.get(oa.oaCdbaNS()) #=> "bit<0>"
bname.get(oa.oaVerilogNS()) #=> "bit[0]"
```
The `oaName` class is used to wrap any of the `oa*Name` classes.
The type of wrapped name can be found with the `getType()` method call.

**Examples:**

```java
name = oa.oaName(oa.oaCdbaNS(), "foo")
name.getType().getName() #=> "scalarName"

name = oa.oaName("foo[0]")
name.getType().getName() #=> "vectorBitName"

name = oa.oaName("foo[0:7]")
name.getType().getName() #=> "vectorName"  name.get(oa.oaVerilogNS()) #=> "foo[0:7]"
```
Lab 3.3

- Goal - Become familiar with OA Namespaces

- Create a script to:
  - Accept a single argument from the command line
  - Store it as a CDBA oaName (oaCdbaNS)
  - Show the oaName type
  - Show the unpacked string in the following namespaces:
    - Spice (oaSpiceNS)
    - Verilog (oaVerilogNS)
    - LEF (oaLefNS)

- Try using various inputs to see if they fail
  - ‘foo’, ‘bit<0>’, ‘bit<0:7>’, ‘a`b’

- Compare your script with labs/3.3/names.py
Section 3 Summary

- OpenAccess models just about everything as an object
- Geometric objects: point, point array, box
- Geometric actions: transform, comparisons (contains, overlaps)
- Names and namespaces
- Some types: not comprehensive, there are more
Silicon Integration Initiative

www.si2.org

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