Java support for PicklingTools: New as of PicklingTools 1.5.0

As of PicklingTools 1.5.0, there is support for Java: this means that Java can handle Python dictionaries and pickle them into files or sockets.

1. What does it means that PicklingTools 1.5.0 supports Java?

   Short Answer: In general, Java is supported like C++, but since the Java baseline isn't quite as fleshed out as the C++ baseline, there are some missing features.

   Long Answer.

a. Java can talk to MidasServers using a Java MidasTalker (i.e., we can have a Java client talking to any C++/Python MidasServer or M2k OpalPythonDaemon). There is currently no support for Java MidasServers, Midasyellers or MidasListeners, but that will be available in a future release. MidasTalker simple example:

```
MidasTalker mt = new MidasTalker("localhost", 8888);
mt.open();
mt.send(new Tab("{'a':1, 'b':2.2, 'c':'three'}");
Object o = mt.recv(5.0);  // block upto 5 seconds waiting
Tab result = (Tab)o;
result.prettyPrint(System.out);  // print as Python dict
```

b. Java can read Python dictionaries from files or sockets. See below for examples with textual and binary (pickled) data.

c. Java can read/write pickled data.

d. Java can read/write textual Python dictionaries. Example:

```
import pythonesque.*;

// Create text dictionary from a Python compatible string
Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'arr':[1,2.2,'t']}");

// Write a textual Python dict to a file
Ptools.WritePythonTextFile("python_dict.txt", t);

// Read a text Python dictionary from a file
Object o = Ptools.ReadPythonTextFile("python_dict.txt");
Tab res = (Tab)o;

// Write a pretty printed textual Python dict to output
res.prettyPrint(System.out);

// Write python dict to output (no extra spaces)
System.out.println(res);
```

e. Java can manipulate Python-esque dict and lists. Example:

```
Tab t = new Tab("{'a':1, 'nest':{'b':2.2, 'c':'three'}");
t.put("newkey", 16);          // Insert new key at top level
t.put("nest", "newer", 2.2);  // Cascading insert into nested dict

Arr a = new Arr("[10000, 2.2, 'three', [1,2,3]]");
int ii = (Integer)a.get(1);     // get int
int ii = (Integer)a.get(3, 0);  // get nested int
```

2. Where does the Pickling support come from?

   The Pyro project released an OpenSource pickling package which PicklingTools 1.5.0 has embraced and is working with. The Pyro project has a similar license as the PicklingTools.

3. How do you handle Python types in Java?

Directly from the Pyro documentation:

```
Pyrolite does the following type mappings:
```

```
PYTHON    ---->     JAVA
------              ----
None                null
bool                boolean
int                 int
long                long or BigInteger  (depending on size)
string              String
unicode             String
complex             net.razorvine.pickle.objects.ComplexNumber
datetime.date       java.util.Calendar
datetime.datetime   java.util.Calendar
datetime.time       java.util.Calendar
datetime.timedelta  net.razorvine.pickle.objects.TimeDelta
float               double   (float isn't used)
array.array         array of appropriate primitive type (char, int, short, long, float, double)
list                java.util.List<Object>
tuple               Object[]
set                 java.util.Set
dict                java.util.Map
bytes               byte[]
bytearray           byte[]
```

The unpickler simply returns an Object. Because Java is a statically typed language you will have to cast that to the appropriate type. Refer to this table to see what you can expect to receive.:

```
JAVA     ---->      PYTHON
-----               ------
null                None
boolean             bool
byte                int
char                str/unicode (length 1)
String              str/unicode
double              float
float               float
int                 int
short               int
BigDecimal          decimal
BigInteger          long
any array           array if elements are primitive type (else tuple)
Object[]            tuple
byte[]              bytearray
java.util.Date      datetime.datetime
java.util.Calendar  datetime.datetime
Enum                the enum value as string
java.util.Set       set
Map, Hashtable      dict
Vector, Collection  list
Serializable        treated as a JavaBean, see below.
JavaBean            dict of the bean's public properties + __class__ for the bean's type.
```

4. Does the Java support Python Object, dict, and list?

   Short Answer: Yes and No.

   PicklingTools wants to make Java programmers as comfortable with native Java types and tools as much as possible, so choices have been made that tend to fit the Java model. See below.

5. Is there an equivalent for C++ Val?

   Short Answer: No, we use the Java Object instead.

   There is no equivalent Val in Java: the Val in C++ was required because C++ doesn't have a heterogeneous, dynamic container type (as C++ is very statically typed, and the library predates the C++ `any` class). Java *already* has a heterogeneous, dynamic container: the `Object`. Most types inherit from `Object`, and can be cast down from `Object` to the appropriate type easily.

6. Is there a Java equivalent for a Python dict?
   Yes.

By default, a Java Tab "is-a" `HashMap<Object, Object>`. So, Tab inherits all the interface from HashMap. BUT: Tab extends the interface significantly to make the Java Tab feel much more like the Python dict.

7. What does Tab add to `HashMap<String, Object>`?

   Five things.

   1. Less typing. Seriously, which would you rather type?:

      ```
      HashMap<String, Object> m = new HashMap<String, Object>();
      Tab t = new Tab();
      ```

   2. The constructor supports creating a literal from a string, i.e.,:

      ```
      Tab t = new Tab("{'a':1, 'somefloat':2.2, 'nest':{'oo':'str'} }");
      // Constructs same table as above
      ```

      This is exactly the syntax of dictionary literals in Python, so these tables can be cut-and-pasted between Python and Java.

   3. Supports pretty print that looks EXACTLY like Python dictionaries so you can cut and paste Python dicts between Python and Java:

      ```
      t.prettyPrint(System.out);
      // overrides toString to System.out.println(t) also works
      ```

   4. Support for cascading lookup:

      ```
      Tab t = new Tab("{'a':1, 'somefloat':2.2, 'nest':{'oo':'str'} }");
      String s = (String)t.get("nest", "oo");
      ```

   5. Support for cascading insertion:

      ```
      Tab t = new Tab("{'nest':{ 'nest2':{} }");
      t.put("nest", "nest2", "value");
      ```

8. How do Java Tab interactions compare with Python dicts?

   They are similar in many ways. Newer features of Java (such as boxing and unboxing) make it a little easier to get stuff in and out of Tabs. For example, In Python:

   ```
   >>> a = { 'a': 1 }
   >>> a['b'] = 17.7
   ```

   The equivalent in Java:

   ```
   Tab a = new Tab("{'a':1}");
   a.put("b", 17.7);             // Because of boxing, don't have to type
                                 // a.put("b", new Double(17.7));
   ```

   Overloading and variable number of arguments make it easier to represent cascading lookups and inserts. In Python:

```
>>> t = {'nest':{'a':1} }
>>> lookup =  t['nest']['a']    # Cascading Lookup
>>> print lookup                # output: 1

>>> t['nest']['a'] = 777        # Cascading insert
>>> print t                     # output: { 'nest': {'a': 777} }
```

In Java:

```
Tab t = new Tab("{'nest': {'a':1}}");
Object lookup = t.get("nest", "a");   // Cascading lookup
System.out.println(lookup);           // output: 1

t.put("nest", "a", 777);              // Cascading insertin
System.out.println(t);                // output: { 'nest': {'a': 777 } }
```

Note that Java uses double quotes ("") for strings and Python can use both single and double quotes ("", '') for strings. In general, it's easier to type single quotes inside the literal string of the Tab.

9. How do we get types out of Java?

   Use the type-casting of Java. For example, to get a nested Tab from another Tab:

   ```
   Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'nest':{} }");
   Object o = t.get("nest");
   Tab nest = (Tab)o;
   ```

   Or, a little less typing:

   ```
   Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'nest':{} }");
   Tab nest = (Tab)t.get("nest");
   ```

   With unboxing, getting POD types like int and floats out isn't quite perfect: they have to go through the Object version. For example:

   ```
   int i = (int)t.get("a");      // SYNTAX ERROR: too much unboxing to do

   int i = (Integer)t.get("a");  // Okay, unboxing helps
   ```

   So, primitive types can work, but do require a cast. (C++ gets around this extra cast because C++ supports a language feature called user-defined conversion). This isn't ideal, but it is a reasonably small amount of typing.

10. Is there an equivalent for a Python list?

    Yes.

    By default, an Arr "is-a" `ArrayList<Object>`. So, Arr inherits all the interface from ArrayList. BUT: Arr extends the interface significantly to make the Java Arr feel much more like the Python list.

11. What does Arr add to `ArrayList<Object>`?

    Five Things. The Arr adds some features to the ArrayList that make it easier to manipulate. The Arr "is-a" ArrayList<Object> so it still supports all the same methods, as well as:

1. Less typing:

```
ArrayList<Object> a = new ArrayList<Object>();
Arr a = new Arr();
```

2. Support for string literals, where string literals can be cut-and-paste directly between Java and Python:

```
Arr a = new Arr("[1, 2.2, 'three', [44]]");
```

3. Supports pretty print that looks EXACTLY like Python dictionaries so you can cut and paste Python dicts between Python and Java:

```
Arr a = new Arr("[1, 2.2, 'three', [44]]");
a.prettyPrint(System.out);
// overrides toString to System.out.println(a) also works
```

4. Supports cascading lookups in nested Arr/Tab:

```
a.get(3, 0);
//  --> gets 44 from nested array
```

5. Supports cascading inserts into nested Arr/Tab:

```
a.put(3, 0, 777);
// --> results in [1, 2.2, 'three', [777]]
```

12. How do Java Arr interactions compare with Python lists?

Like Tabs, boxing and unboxing make it easier to deal with heterogeneous types in Java. Consider a Python list:

```
>>> a = [1, 2.2, 'three']
>>> a.append(6)
```

Similarly in Java:

```
Arr a = new Arr("[1, 2.2, 'three']");
a.add(6);        // Because of boxing, don't have to do
                 //    a.add(new Integer(6));
```

Overloading get and put, along with Java supporting a variable number of arguments (as well as boxing/unboxing) make cascading gets and puts (that Python deals with so easily) easy to deal with. Consider Python:

```
>>> aa = [ 5, 6, [10, 11] ]
>>> lookup =  aa[2][0]     # Cascading lookup: 10
>>> print lookup

>>> aa[2][0] = 100     # Cascading inserts: [5,6, [100,11]
>>> print aa
```

In Java, the equivalent is:

```
Arr aa = new Arr("[5,6, [10,11]]");
Object lookup = aa.get(2,0);    // Cascading lookup
System.out.println(lookup);

aa.put(2, 0, 100);              // Cascading insert: [5,6, [100,11]
System.out.println(aa);
```

13. Is there support for Tuples and OTabs?

    Tuples: yes, limited. This will be expanded in a later version. OTab. No. May be added later.

14. What's missing?

    There's still some development to do.

    1. No MidasServer. This is currently no MidasServer like in C++/Python there is only the client.
    2. No Numeric or NumPy support. The original Pyrolite dist. only supported Python Arrays.
    3. No "shared" data structures. The pytolite dist. did not implement the get/put feature of pickling which allows dictionaries to be shared (i.e., only deep copies are made).
    4. No Unicode support. The C++/Python tools were developed to work with Python 2.x series, where strings where ASCII. The unicode support for PTOOLS is non-existant currently: this is a PTOOLS problem, not a Java problem.
    5. No byte array support. It becomnes unicode, which (4) can't handle.
    6. Can't read textual arrays. Cannot currently read arrays from strings or files.
    7. No MidasYeller or MidasListener.

    These are the major missing pieces, which we will flesh out as people need them. This is a first release to get the basic functionality out the door so we can get feedback.