JPype Documentation

Release 0.7.5

Steve Menard, Luis Nell and others

Aug 07, 2020

Contents

JPype is a Python module to provide full access to Java from within Python. It allows Python to make use of Java specific libraries, explore and visualize Java structures, develop and test Java libraries, make use of scientific computing, and much more. By enabling the use of Python for rapid prototyping and Java for strong typed production code, JPype provides a powerful environment for engineering and code development.

Unlike Jython, JPype does not achive this by re-implementing Python, but instead by interfacing both virtual machines at the native level. This shared memory based approach achieves good computing performance, while providing the access to the entirety of CPython and Java libraries.

CHAPTER 1

Parts of the documentation

1.1 Installation

JPype is available either as a pre-compiled binary for Anaconda, or may be built from source though various methods.

1.1.1 Binary Install

JPype can be installed as pre-compiled binary if you are using the Anaconda Python stack. Binaries are available for Linux, OSX, and windows on conda-forge.

1. Ensure you have installed Anaconda/Miniconda. Instructions can be found here.
2. Install from the conda-forge software channel:

```
conda install -c conda-forge jpype1
```

1.1.2 Source Install

Installing from source requires:

Python JPype works CPython 3.5 or later. Both the runtime and the development package are required.

Java Either the Sun/Oracle JDK/JRE Variant or OpenJDK.

JPype source distribution includes a copy of the Java JNI header and precompiled Java code, thus the Java Development Kit (JDK) is not required. JPype has been tested with Java versions from Java 1.7 to Java 13.

C++ A C++ compiler which matches the ABI used to build CPython.

JDK *(Optional)* JPype contains sections of Java code. These sections are precompiled in the source distribution, but must be built when installing directly from the git repository.

Once these requirements have been met, one can use pip to build from either the source distribution or directly from the repository. Specific requirements from different achitectures are listed *below*.

Build using pip

JPype may be built and installed with one step using pip.

To install the latest JPype, use:

```
pip install JPype1
```

This will install JPype either from source or binary distribution, depending on your operating system and pip version.

To install from the current github master use:

```
pip install git+https://github.com/jpype-project/jpype.git
```

More details on installing from git can be found at Pip install. The git version does not include a prebuilt jar the JDK is required.

Build and install manually

JPype can be built entirely from source.

1. Get the JPype source

The JPype source may be acquired from either github or from PyPi.

2. Build the source with desired options

Compile JPype using the included `setup.py` script:

```
python setup.py build
```

The setup script recognizes several arguments.

--enable-build-jar Force setup to recreate the jar from scratch.

--enable-tracing Build a verison of JPype with full logging to the console. This can be used to diagnose tricky JNI issues.

After building, JPype can be tested using the test bench. The test bench requires JDK to build.

3. Test JPype with (optional):

```
python setup.py test
```

4. Install JPype with:

```
python setup.py install
```

If it fails...

Most failures happen when setup.py is unable to find the JDK home directory which shouble be set in the enviroment variable `JAVA_HOME`. If this happens, preform the following steps:

1. Identify the location of your systems JDK installation and explicitly passing it to setup.py.

   ```
   JAVA_HOME=/usr/lib/java/jdk1.8.0/ python setup.py install
   ```

2. If that setup.py still fails please create an Issue on github and post the relevant logs.

Platform Specific requirements

JPype is known to work on Linx, OSX, and Windows. To make it easier to those who have not built CPython modules before here are some helpful tips for different machines.

Debian/Ubuntu

Debian/Ubuntu users will have to install `g++` and `python-dev`. Use:

```
sudo apt-get install g++ python-dev python3-dev
```

Windows

CPython modules must be built with the same C++ compiler used to build Python. The tools listed below work for Python 3.5 to 3.8. Check with Python dev guide for the latest instructions.

1. Install your desired version of Python (3.5 or higher), e.g., Miniconda is a good choice for users not yet familiar with the language
2. For Python 3 series, Install either 2017 or 2019 Visual Studio. Microsoft Visual Studio 2019 Community Edition is known to work.

From the Python developer page:

> When installing Visual Studio 2019, select the Python development workload and the optional Python native development tools component to obtain all of the necessary build tools. If you do not already have git installed, you can find git for Windows on the Individual components tab of the installer.

When building for windows you must use the Visual Studio developer command prompt.

1.1.3 Path requirements

On certain systems such as Windows 2016 Server, the JDK will not load properly despite JPype properly locating the JVM library. The work around for this issue is add the JRE bin directory to the system PATH. Apparently, the shared library requires dependencies which are located in the bin directory. If a JPype fails to load despite having the correct JAVA_HOME and system architecture, it may be this issue.

1.1.4 Known Bugs/Limitations

- Java classes outside of a package (in the `<default>`) cannot be imported.
- Because of lack of JVM support, you cannot shutdown the JVM and then restart it. Nor can you start more than one copy of the JVM.
- Mixing 64 bit Python with 32 bit Java and vice versa crashes on import of the jpype module.

1.2 JPype User Guide

1.2.1 JPype Introduction

JPype is a Python module to provide full access to Java from within Python. Unlike Jython, JPype does not achive this by re-implementing Python, but instead by interfacing both virtual machines at the native level. This shared memory

based approach achieves good computing performance, while providing the access to the entirety of CPython and Java libraries. This approach allows direct memory access between the two machines, implementation of Java interfaces in Python, and even use of Java threading.

JPype Use Cases

Here are three typical reasons to use JPype.

- Access to a Java library from a Python program (Python oriented)
- Visualization of Java data structures (Java oriented)
- Interactive Java and Python development including scientific and mathematical programming.

Let's explore each of these options.

Case 1: Access to a Java library

Suppose you are a hard core Python programmer. You can easily use lambdas, threading, dictionary hacking, monkey patching, been there, done that. You are hard at work on your latest project but you just need to pip in the database driver for your customers database and you can call it a night. Unfortunately, it appears that your customers database will not connect to the Python database API. The whole thing is custom and the customer isn't going to supply you with a Python version. They did sent you a Java driver for the database but fat lot of good that will do for you.

Stumbling through the internet you find a module that says it can natively load Java packages as Python modules. Well, it worth a shot...

So first thing the guide says is that you need to install Java and set up a `JAVA_HOME` environment variable pointing to the JRE. Then start the JVM with classpath pointed to customers jar file. The customer sent over an example in Java so you just have to port it into Python.

```
package com.paying.customer;

import com.paying.customer.DataBase

public class MyExample {
   public void main(String[] args) {
     Database db = new Database("our_records");
     try (DatabaseConnection c = db.connect())
     {
        c.runQuery();
        while (c.hasRecords())
        {
          Record record = db.nextRecord();
          ...
        }
     }
  }
}
```

It does not look too horrible to translate. You just need to look past all those pointless type declarations and meaningless braces. Once you do, you can glue this into Python and get back to what you really love, like performing dictionary comprehensions on multiple keys.

You glance over the JPype quick start guide. It has a few useful patterns... set the class path, start the JVM, remove all the type declarations, and you are done.

```
# Boiler plate stuff to start the module
import jpype
import jpype.imports
from jpype.types import *

# Launch the JVM
jpype.startJVM(classpath=['jars/database.jar'])

# import the Java modules
from com.paying.customer import DataBase

# Copy in the patterns from the guide to replace the example code
db = Database("our_records")
with  db.connect() as DatabaseConnection:
   c.runQuery()
   while c.hasRecords():
       record = db.nextRecord()
       ...
```

Launch it in the interactive window. You can get back to programming in Python once you get a good night sleep.

Case 2: Visualization of Java structures

Suppose you are a hard core Java programmer. Weakly typed languages are for wimps, if it isn't garbage collected it is garbage. Unfortunately your latest project has suffered a nasty data structure problem in one of the threads. You managed to capture the data structure in a serialized form but if you could just make graph and call a few functions this would be so much easier. But the interactive Java shell that you are using doesn't really have much in the way of visualization and your don't have time to write a whole graphing applet just to display this dataset.

So poking around on the internet you find that Python has exactly the visualization that you need for the problem, but it only runs in CPython. So in order to visualize the structure, you need to get it into Python, extract the data structures and, send it to the plotting routine.

You install conda, follow the install instructions to connect to conda-forge, pull JPype1, and launch the first Python interactive environment that appear to produce a plot.

You get the shell open and paste in the boilerplate start commands, and load in your serialized object.

```
import jpype
import jpype.imports

jpype.startJVM(classpath = ['jars/*', 'test/classes'])

from java.nio.file import Files, Paths
from java.io import ObjectInputStream

with Files.newInputStream(Paths.get("myobject.ser") as stream:
   ois = new ObjectInputStream(stream)
   obj = ois.readObject()

print(obj) # prints org.bigstuff.MyObject@7382f612
```

It appears that the structure is loaded. The problematic structure requires you call the getData method with the correct index.

```
d = obj.getData(1)

> TypeError: No matching overloads found for org.bigstuff.MyObject.getData(int),
> options are:
       public double[] org.bigstuff.MyObject.getData(double)
       public double[] org.bigstuff.MyObject.getData(int)
```

Looks like you are going to have to pick the right overload as it can't figure out which overload to use. Darn weakly typed language, how to get the right type in so that you can plot the right data. It says that you can use the casting operators.

```
from jpype.types import *
d = obj.getData(JInt(1))
print(type(d))  # prints double[]
```

Great. Now you just need to figure out how to convert from a Java array into our something our visualization code can deal with. As nothing indicates that you need to convert the array, you just copy out of the visualization tool example and watch what happens.

```
import matplot.pyplot as plt
plt.plot(d)
plt.show()
```

A graph appears on the screen. Meaning that NumPy has not issue dealing with Java arrays. It looks like ever 4th element in the array is zero. It must be the PR the new guy put in. And off you go back to the wonderful world of Java back to the safety of curly braces and semicolons.

Case 3: Interactive Java

Suppose you are a laboratory intern running experiments at Hawkins National Laboratory. (For the purpose of this exercise we will ignore the fact that Hawkins was shut down in 1984 and Java was created in 1995). You have the test subject strapped in and you just need to start the experiment. So you pull up Jupyter notebook your boss gave you and run through the cells. You need to added some heart wave monitor to the list of graphed results.

The relevant section of the API for the Experiment appears to be

```
package gov.hnl.experiment;

public interface Monitor {
   public void onMeasurement(Measurement measurement);
}

public interface Measurement {
   public double getTime();
   public double getHeartRate();
   public double getBrainActivity();
   public double getDrugFlowRate();
   public boolean isNoseBleeding();
}

public class Experiment {
   public void addCondition(Instant t, Condition c);
   public void addMoniter(Monitor m);
   public void run();
}
```

The notebook already has all the test conditions for the experiment set up and the JVM is started, so you just need to implement the monitor.

Based on the previous examples, you start by defining a monitor class

```
from jpype import JImplements, JOverride
from gov.hnl.experiment import Monitor

@JImplements(Monitor)
class HeartMonitor:
   def __init__(self):
      self.readings = []
   @JOverride
   def onMeasurement(self, measurement):
      self.readings.append([measurement.getTime(), measurement.getHeartRate()])
   def getResults(self):
      return np.array(self.readings)
```

There is a bit to unpack here. You have implemented a Java class from within Python. The Java implementation is simply an ordinary Python class which has be decorated with `@JImplements` and `@JOverride`. When you forgot to place the `@JOverride`, it gave you the response:

```
NotImplementedError: Interface 'gov.hnl.experiment.Monitor' requires
method 'onMeasurement' to be implemented.
```

But once you added the `@JOverride`, it worked properly. The subject appears to be getting impatient so you hurry up and set up a short run to make sure it is working.

```
hm = HeartMonitor()
experiment.addMonitor(hm)
experiment.run()
readings = hm.getResults()
plt.plot(readings[:,0], readings[:,1)
plt.show()
```

To your surprise, it says unable to find method addMonitor with an error message:

```
AttributeError: 'gov.hnl.experiment.Experiment' object has no attribute 'addMonitor'
```

You open the cell and type `experiment.add<TAB>`. The line completes with `experiment.addMoniter`. Whoops, looks like there is typo in the interface. You make a quick correction and see a nice plot of the last 30 seconds pop up in a window. Job well done, so you set the runtime back to one hour. Looks like you still have time to make the intern woodlands hike and forest picnic. Though you wonder if maybe next year you should sign up for another laboratory. Maybe next year, you will try to sign up for those orbital lasers the President was talking about in the March. That sounds like real fun.

(This advanced demonstration utilized the concept of *Proxies* and *Code completion*)

The JPype Philosophy

JPype is designed to allow the user to exercise Java as fluidly as possible from within Python. We can break this down into a few specific design goals.

- Make Java appear Pythonic. Make it so a Python programmer feels comfortable making use of Java concepts. This means making use of Python concepts to create very Python looking code and at times bending Python concepts to conform to Java's expectations.

- Make Python appear like Java. Present concepts from Java with a syntax that resembles Java so that Java users can work with Python without a huge learning curve.
- Present everything that Java has to offer to Python. Every library, package, and Java feature if possible should be accessible. The goal of bridge is to open up places and not to restrict flow.
- Keep the design as simple as possible. Mixing languages is already complex enough so don't required the user to learn a huge arsenal of unique methods. Instead keep it simple with well defined rules and reuse these concepts. For example, all array types originate from JArray, and thus using one can also use isinstance to check if a class is an array type. Rather than introducing factory that does a similar job to an existing one, instead use a keyword argument on the current factory.
- Favor clarity over performance. This doesn't mean not trying to optimize paths, but just as premature optimization is the bane of programmers, requiring writing to maximize speed is a poor long term choice, especially in a language such as Python were weak typing can promote bit rot.
- If a new method has to be introduced, make look familiar. Java programmers look to a method named "of" to convert to a type on factories such as a Stream, thus `JArray.of` converts a Python NumPy array to Java. Python programmers expect that memory backed objects can be converted into bytes for rapid transfer using a memory view, thus `memoryview(array)` will perform that task.
- Provide an obvious way for both Python and Java programmers to perform tasks. On this front JPype and Python disagree. In Python's philosophy there should be one – and preferably only one – obvious way to do things. But we are bridging two worlds and thus obviousness is in the eye of the beholder.

The end result is that JPype has a small footprint while providing access to Java (and other JVM based languages) with a minimum of effort.

Languages other than Java

JPype is primarily focused on providing the best possible wrapper for Java in Python. However, the Java Virtual Machine (JVM) is used for many popular languages such a Kotlin and Scala. As such JPype can be used for any language which used the JVM.

That said each language has its own special properties that tend to be represented in different ways. If you would like JPype fully to operate on your particular language the following is required.

- Set up a test bench for your language under the test directory. Use ivy to pull in the required jar files required to run it and exercise each of the required language features that need to be exercised.
- Write a language specific quick start guide for your language defining how things should appear in both your language of choice and within Python highlighting those things that are different from how Java.
- Set up a test harness that exercises your language for each language feature and place a setup script like `test_java` that builds the harness.

Alternatives

JPype is not the only Python module of its kind that acts as a bridge to Java. Depending on your programming requirements, one of the alternatives may be a better fit. Specifically JPype is designed for clarity and high levels of integration between the Python and Java virtual machine. As such it makes use of JNI and thus inherits all of the benefits and limitations that JNI imposes. With JPype, both virtual machines are running in the same process and are sharing the same memory space and threads. JPype can thus intermingle Python and Java threads and exchange memory quickly. But by extension you can't start and stop the JVM machine but instead must keep both machines throughout the lifespan of the program. High integration means tightly coupled and thus it embodies the musketeers motto. If Python crashes, so does Java as they only have one process to live in.

A few alternatives with different philosophies and limitations are given in the following section. Please take my review comments with the appropriate grain of salt. When I was tasked with finding a replacement for Matlab Java integration for our project test bench, I evaluated a number of alternatives Python bridge codes. I selected JPype primarily because it presented the most integrated API and documentation which would be suitable for getting physicists up to speed quickly. Thus your criteria may yield a different selection. Its underlying technology was underwhelming and thus I have had the pleasure of many hours reworking stuff under the hood.

For more details on what you can't do with JPype, please see *Limitations*.

Jython

Jython is a reimplementation of Python in Java. As a result it has much lower costs to share data structures between Java and Python and potentially much higher level of integration. Noted downsides of Jython are that it has lagged well behind the state of the art in Python; it has a limited selection of modules that can be used; and the Python object thrashing is not particularly well fit in Java virtual machine leading to some known performance issues.

Py4J

Py4J uses a remote tunnel to operate the JVM. This has the advantage that the remote JVM does not share the same memory space and multiple JVMs can be controlled. It provides a fairly general API, but the overall integration to Python is as one would expect when operating a remote channel operating more like an RPC front-end. It seems well documented and capable. Although I haven't done benchmarking, a remote access JVM will have a transfer penalty when moving data.

Jep

Jep stands for Java embedded Python. It is a mirror image of JPype. Rather that focusing on accessing Java from within Python, this project is geared towards allowing Java to access Python as sub-interpreter. The syntax for accessing Java resources from within the embedded Python is quite similar with support for imports. Notable downsides are that although Python supports multiple interpreters many Python modules do not, thus some of the advantages of the use of Python many be hard to realize. In addition, the documentation is a bit underwhelming thus it is difficult to see how capable it is from the limited examples.

Javabridge

Javabridge is direct low level JNI control from Python. The integration level is quite low on this, but it does serve the purpose of providing the JNI API to Python rather than attempting to wrap Java in a Python skin. The downside being of course you would really have to know a lot of JNI to make effective use of it.

jpy

This is the most similar package to JPype in terms of project goals. They have achieved more capabilities in terms of a Java from Python than JPype which does not support any reverse capabilities. It is currently unclear if this project is still active as the most recent release is dated 2014. The integration level with Python is fairly low currently though what they do provide is a similar API to JPype.

About this guide

The JPype User Guide is targeted toward programmers who are strong in either Python who wish to make use of Java or those who are strong with Java and are looking to use Python as a Java development tool. As such we will compare and contrast the differences between the languages and provide examples suitable to help illustrate how to translate from one language to the other on the assumption that being strong in one language will allow you to easily grasp the corresponding relations in the other. If you don't have a strong background in either language an appropriate language tutorial may be necessary.

JPype will hide virtually all of the JNI layer such that there is no direct access to JNI concepts. As such attempting to use JNI knowledge will likely lead to incorrect assumptions such as incorrectly attempting to use JNI naming and method signatures in the JPype API. Where JNI limitations do appear we will discuss the consequences imposed in programming. No knowledge of JNI is required to use this guide or JPype.

JPype only works with Python 3, thus all examples will be using Python version 3 syntax and assume the use of the Python 3 new style object model. The naming conventions of JPype follow the Java rules rather than those of Python. This is a deliberate choice as it would be dangerous to try to mangle Java method and field names into Python conventions and risk a name collision. Thus if method must have Java conventions then the rest of the module should follow the same pattern for consistency.

Getting JPype started

This document holds numerous JPype examples. For the purposes of clarity the module is assumed to have been started with the following command

```
# Import the module
import jpype

# Allow Java modules to be imported
import jpype.imports

# Import all standard Java types into the global scope
from jpype.types import *

# Import each of the decorators into the global scope
from jpype import JImplements, JOverride, JImplementationFor

# Start JVM with Java types on return
jpype.startJVM(convertStrings=False)

# Import default Java packages
import java.lang
import java.util
```

This is not the only style used by JPype users. Some people feel it is best to limit the number for symbols in the global scope and instead start with a minimalistic approach.

```
import jpype as jp                  # Import the module
jp.startJVM(convertStrings=False)   # Start the module
```

Either style is usable and we do not wish to force any particular style on the user. But as the extra `jp.` tends to just clutter up the space and implies that JPype should always be used as a namespace due to namespace conflicts, we have favored the global import style. JPype only exposes 40 symbols total including a few deprecated functions and classes. The 13 most commonly used Java types are wrapped in a special module `jpype.types` which can be used to import all for the needed factories and types with a single command without worrying about importing potentially problematic symbols.

We will detail the starting process more later in the guide. See *Starting the JVM*.

1.2.2 JPype Concepts

At its heart, JPype is about providing a bridge to use Java within Python. Depending on your prospective that can either be a means of accessing Java libraries from within Python or a way to use Java using Python syntax for interactivity and visualization. This mean not only exposing a limited API but instead trying to provide the entirety of the Java language with Python.

To do this, JPype maps each of the Java concepts to the nearest concept in Python wherever they are similar enough to operate without confusion. We have tried to keep this as Pythonic as possible, though it is never without some rough edges.

Python and Java share many of the same concepts. Types, class, objects, function, methods, and members. But in other places they are rather different. Python lacks casting, type declarations, overloading, and many other features of a strongly typed language, thus we must expose those concepts into the Python syntax as best we can. Java for instance has class annotation and Python have class decorators. Both serve the purpose of augmenting a class with further information, but are very different in execution.

We have broken the mapping down in nine distinct concepts. Some elements serve multiple functions.

Type Factories These are meta classes that allow one to declare a particular Java type in Python. The result of type factories are wrapper classes. (*JClass* and *JArray*) Factories also exist to implement Java classes from within Python (*JProxy*)

Meta Classes These are classes to describe different properties of Java classes such as to check if a class is an Interface. (*JInterface*)

Base Classes These are JPype names for Java classes in Python that exist without importing any specific Java class. Concepts such as Object, String, and Exception are defined and can be used in instance checks. For example, to catch all Java exceptions regardless of type, we would catch `JException`. These are mainly for convenience though they do have some extra functionality. Most of these functions are being phased out in favor of Java syntax. For example, catching `java.lang.Throwable` will catch everything that `JException` will catch. (*Jarray*, *JObject*, *JString*, and *JException*)

Wrapper Classes These correspond to each Java class. Thus can be used to access static variables, static methods, cast, and construct object. They are used wherever a Java type would be used in the Java syntax such as creating an array or accessing the class instance. These class wrappers are customized in Python to allow a direct mapping from Java concepts to Python one. These are all created dynamically corresponding to each Java class. For most of this document we will refer to these simply as a "class". (*java.lang.Object*, *java.lang.String*, etc) Many wrappers are customized to match Python abstract base classes ABC (*java.util.List*, *java.util.Map*)

Object Instances These are Java objects. They operate just like Python objects with Java public fields mapped to Python attributes and Java methods to Python methods. For this document we will refer to an object instance simply as an "object". The object instance is split into two halves. The Python portion is referred to as the "handle" that points the Java "instance". The lifetime of the "instance" is tied to the handle thus Java objects do not disappear until the Python handle is disposed of. Objects can be *cast* to match the required type and hold *methods* and fields.

Primitive types Each of the 8 Java primitive types are defined. These are used to cast to a Java type or to construct arrays. (*JBoolean*, *JChar*, *JByte*, *JShort*, *JInt*, *JLong*, *JFloat*, and *JDouble*)

Decorators Java has a number of keywords such as extending a class or implementing an interface. Those pieces of meta data can't directly be expressed with the Python syntax, but instead have been been expressed as annotations that can be placed on classes or functions to augment them with Java specific information. (*@JImplements*, *@JOverride*, *@JImplementationFor*)

Mapping Java syntax to Python Many Java concepts like try with resources can be mapped into Python directly (as the `with` statement), or Java try, throw, catch mapping to Python try, raise, except. Others such as synchronize

do not have an exact Python match. Those have instead been mapped to special functions that interact with Python syntax.. (*synchronized*, *with*, *try*, *import*)

JVM control functions The JVM requires specific actions corresponding to JNI functions in order to start, shutdown, and define threading behavior. These top level control functions are held in the `jpype` module. (*startJVM*, *shutdownJVM*)

We will detail each of these concepts in greater detail in the later sections.

Name mangling

When providing Java package, classes, methods, and fields to Python, there are occasionally naming conflicts. For example, if one has a method called `with` then it would conflict with the Python keyword `with`. Wherever this occurs, JPype renames the offending symbol with a trailing under bar. Java symbols with a leading or trailing under bars are consider to be privates and may not appear in the JPype wrapper entirely with the exception of package names.

The following Python words will trigger name mangling of a Java name:

`False`	`None`	`True`	`and`	`as`
`async`	`await`	`def`	`del`	`elif`
`except`	`exec`	`from`	`global`	`in`
`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`print`	`raise`	`with`	`yield`

1.2.3 JPype Types

Both Java and Python have a concept of a type. Every variable refers to an object which has a defined type. A type defines the data that the variable is currently holding and how that variable can be used. In this chapter we will learn how Java and Python types relate to one another, how to create import types from Java, and how to use types to create Java objects.

Stay strong in a weak language

Before we get into the details of the types that JPype provides, we first need to contrast some of the fundamental language differences between Java and Python. Python is inherently a weakly typed language. Any variable can take any type and the type of a particular variable can change over the lifetime of a program. Types themselves can be mutable as you can patch an existing type to add new behaviors. Python methods can in principle take any type of object as an argument, however if the interface is limited it will produce a TypeError to indicate a particular argument requires a specific type. Python objects and classes are open. Each class and object is basically a dictionary storing a set of key value pairs. Types implemented in native C are often more closed and thus can't have their method dictionaries or data members altered arbitrarily. But subject to a few restrictions based implementation, it is pretty much the wild west.

In contrast, Java is a strongly typed language. Each variable can only take a value of the specified class or a class that derives from the specified class. Each Java method takes only a specific number and type of arguments. The type and number are all checked at compile type to ensure there is little possibility of error. As each method requires a specific number and type of arguments, a method can be overloaded by having two different implementations which take a different list of types sharing the same method name. A primitive variable can never hold an object and it can only be converted to or from other primitive types unless it is specifically cast to that type. Java objects and classes are completely closed. The methods and fields for a particular class and object are defined entirely at compile time. Though it is possible create classes with a dictionary allowing expansion, this is not the Java norm and no standard mechanism exists.

Thus we need to introduce a few Java terms to the Python vocabulary. These are "conversion" and "cast".

Java conversions

A conversion is a permitted change from an object of one type to another. Conversions have three different degrees. These are: exact, implicit, and explicit.

Exact conversions are those in which the type of an object is identical. In Java each class has only one definition thus there is no need for an exact conversion. But when dealing with Python we have objects that are effectively identical for which exact conversion rules apply. For example, a Java string and a Python string both bind equally well to a method which requires a string, thus this is an exact conversion for the purposes of bind types.

The next level of conversion is implicit. An implicit conversion is one that Java would perform automatically. For example converting a derived class to is base class when setting a field would be an implicit conversion. Java defines a number of other conversions such as converting a primitive to a boxed type or from a boxed type back to a primitive as implicit conversions..

Of course not every cast is safe to perform. For example, converting an object whose type is currently viewed as a base type to a derived type is not performed automatically nor is converting from one boxed type to another. For those operations the conversion must be explicitly requested, hence these are explicit conversions. To request an explicit conversion an object must be "cast" using a cast operator. In Java, a cast is requested by placing the type name in parentheses in front of the object to be cast. Unfortunately, the same syntax is not allowed in Python. Not every conversion is possible between Java types. Types that cannot be converted are considerer to be conversion type "none".

Details on the standard conversions provided by JPype are given in the section *Type Matching*.

Java casting

To access a casting operation we use the casting `JObject` wrapper. JObject accepts two arguments. The first argument is the object to convert and the second is the type to cast to. The second argument should always be a Java type specified using a class wrapper, a Java class instance, or a string. Casting will also add a hidden class argument to the resulting object such that it is treated as the cast type for the duration of that variable lifespan. Therefore, a variable create by casting is stuck as that type and cannot revert back to its original for the purposes of method resolution.

The object construction and casting are sometimes a bit blurry. For example, when one casts a sequence to a Java list, we will end up constructing a new Java list that contains the elements of the original Python sequence. In general JPype constructors only provide access the Java constructor methods that are defined in the Java documentation. Casting on the other hand is entirely the domain of whatever JPype has defined including user defined casts.

Casting is performed through the Python class `JObject`. JObject is called with two arguments which are the object to be cast and the type to cast too. The cast first consults the conversion table to decide if the cast it permitted and produces a `TypeError` if the conversion is not possible.

`JObject` also serves as a abstract base class for testing if an object instance belongs to Java. All objects that belong to Java will return true when tested with `isinstance`. Like Python's sequence, JObject is an abstract base class. No classes actual derive from `JObject`.

Of particular interest is the concept of Java `null`. In Java, null is a typeless entity which can be placed wherever an object is taken to indicate that the object is not available. The equivalent concept in Python is `None`. Thus all methods that accept any object type that permit a null will accept None as an augment with implicit conversion. However, sometime it is necessary to pass an explicit type to the method resolution. To achieve this in JPype use `JObject(None, type)` which will create a null pointer with the desired type. To test if something is null we have to compare the handle to None. This unfortunately trips up some code quality checkers. The idiom in Python is `obj is None`, but as this only matches things that Python considers identical, we must instead use `obj==None`.

Type enforcement appears in three different places within JPype. These are whenever a Java method is called, whenever a Java field is set, and whenever Python returns a value back to Java.

Method resolution

Because Java supports method overloading and Python does not, JPype wraps Java methods as a "method dispatch". The dispatch is a collection of all of the methods from class and all of its parents which share the same name. The job of the dispatch is chose the method to call.

Enforcement of the strong typing of Java must be performed at runtime within Python. Each time a method is invoked, JPype must match against the list of all possible methods that the class implements and choose the best possible overload. For this reason the methods that appear in a JPype class will not be the actual Java methods, but rather a "dispatch" whose job is deciding which method should be called based on the type of the provided arguments.

If no method is found that matches the provided arguments, the method dispatch will produce a `TypeError`. This is the exact same outcome that Python uses when enforcing type safety within a function. If a type doesn't match a `TypeError` will be produced.

Dispatch example

When JPype is unable to decide which overload of a method to call, the user must resolve the ambiguity. This is where casting comes in.

Take for example the `java.io.PrintStream` class. This class has a variant of the print and println methods!

So for the following code:

```
java.lang.System.out.println(1)
```

JPype will automatically choose the `println(long)` method, because the Python int matches exactly with the Java long, while all the other numerical types are only "implicit" matches. However, if that is not the version you wanted to call you must cast it. In this case we will use a primitive type to construct the correct type.

Changing the line thus:

```
java.lang.System.out.println(JByte(1)) # <--- wrap the 1 in a JByte
```

This tells JPype to choose the byte version. When dealing with Java types, JPype follows the standard Java matching rules. Types can implicitly grow to larger types but will not shrink without an explicit cast.

Primitive Types

Unlike Python, Java makes a distinction between objects and primitive data types. Primitives represent the minimum data that can be manipulated by a computer. These stand in contrast to objects which have the ability to contain any combination of data types and object within themselves, and can be inherited from.

Java primitives come in three flavors. The logical primitive `boolean` can only take the logical value true and false. The textual primitive `char` represents one character in a string. Numerical primitives are intended for fixed point or floating point calculations. Numerical primitives come in many sizes depending on how much storage is required. In Java, integer numerical primitives are always signed and thus can only reach half their range in terms of bits up or down relative to their storage size.

JPype has mapped each of the primitive types into Python classes. To avoid conflicts with Python, JPype has named each primitive with a capital letter `J` followed by the primitive name starting with an upper case letter.

JBoolean A boolean is the logical primitive as it can only take values `True` and `False`. It should properly be an extension of the Python concept `bool` but that type is not extendable. Thus instead it must inherit from `int`. This type is rarely seen in JPype as the values `True` and `False` are considered an exact match to `JBoolean` argument. Methods which return a `JBoolean` will always return a Python `bool` rather than a Java primitive type.

JChar A character is the textual primitive that corresponds to exactly one character in a string. Or at least that was the concept at the time. Java characters can only represent 16 bits. But there are currently 143,924 defined characters in Unicode. Thus, there are certain characters that can only be represented as two Unicode characters. The textual primitives are not intended to perform numerical functions, but are instead encoded. As per the old joke, what does *1* plus *1* equal? Which of course the correct answer is *b*. As such characters should not be treated as just another unsigned short. Python has no concept of a textual only type. Thus when returning a character type, we instead return a string length 1. The actually `JChar` class is derived from a Python `int` and by inheritance has all of the numerical operations associated with it. There are of course lots of useful mathematical operations that can be performed on textual primitives, but doing so risks breaking the encoding and can result in uninterpretable data.

JByte, Short, Int, Long These types represent fixed point quantities with ranges of 8, 16, 32, and 64 bits. Each of these type inherit from a Python `int` type. A method or field returning an integer primitive will return a type derived from `int`. Methods accepting an integer primitive will take either an Java integer primitive or a Python `int` or anything that quacks like a `int` so long as it can be converted into that primitive range without truncation.

JFloat, JDouble These two types hold floating point and correspond to either single point (32 bit) or double point (64 bit) precision. Python does not have a concept of precision and thus both of these derive from the Python type `float`. As per Java rules numbers greater than the range correspond to the values of positive and negative infinity. Conversions from Python types are ranged check and will produce a `OverflowError` if the value doesn't fit into the request types. If an overflow error is not desired, first cast the value into the request size prior to calling. Methods that return a Java floating point primitive will always return a value derived from `float`.

The classes for Java primitives are closed and should not be extended. As with all Java values any information attached to the Python representation is lost when passing that value to Java.

Objects & Classes

In contrast to primitive data type, objects can hold any combination of primitives or objects. Thus they represent structured data. Objects can also hold methods which operate on that data. Objects can inherit from one another.

However unlike Python, Java objects must have a fixed structure which defines its type. These are referred to the object's class. Here is a point of confusion. Java has two different class concepts: the class definition and the class instance. When you import a class or refer to a method using the class name you are accessing the class definition. When you call `getClass` on an object it returns a class instance. The class instance is a object whose structure can be used to access the data and methods that define the class through reflection. The class instance cannot directly access the fields or method within a class but instead provides its own interface for querying the class. For the purposes of this document a "class" will refer to the class definition which corresponds to the Python concept of a class. Wherever the Java reflection class is being referred to we will use the term "class instance". The term "type" is synonymous with a "class" in Java, though often the term "type" is only used when inclusively discussing the type of primitives and objects, while the term "class" generally refers to just the types associated with objects.

All objects in Java inherit from the same base class `java.lang.Object`, but Java does not support multiple inheritance. Thus each class can only inherit from a single parent. Multiple inheritance, mix-ins, and diamond pattern are not possible in Java. Instead Java uses the concept of an interface. Any Java class can inherit as many interfaces as it wants, but these interfaces may not contain any data elements. As they do not contain data elements there can be no ambiguity as to what data a particular lookup.

The meta class `JInterface` is used to check if a class type is an interface using `isinstance`. Classes that are pure interfaces cannot be instantiated, thus, there is not such thing as an abstract instance. Therefore, every Java object should have Objects cannot actual be pure interfaces. To represent this in Python every interface inherits `java.lang.Object` methods even through it does not have `java.lang.Object` as a parent. This ensures that anonymous classes and lambdas have full object behavior.

Classes

In JPype, Java classes are instances of the Python `type` and function like any ordinary Python class. However unlike Python types, Java classes are closed and cannot be extended. To enforce extension restrictions, all Java classes are created from a special private meta class called `_jpype._JClass`. This gatekeeper ensures that the attributes of classes cannot be changed accidentally nor extended. The type tree of Java is fixed and closed.

All Java classes have the following functionality.

Class constructor The class constructor is accessed by using the Python call syntax `()`. This special method invokes a dispatch whenever the class is called as a function. If an matching constructor is found a new Java instance is created and a Python handle to that instance is returned. In the case of primitive types, the constructor creates a Java value with the exact type requested.

Get attribute The Python `.` operator gets an attribute from a class with a specified name. If no method or field exists a `AttributeError` will be raised. For public static methods, the getattr will produce a Python descriptor which can be called to invoke the static method. For public static fields, a Python descriptor will be produced that allows the field to be get or set depending on whether the field is final or not. Public instance methods and instance fields will produce a function that can be applied to a Java object to execute that method or access the field. Function accessors are non-virtual and thus they can provide access to behaviors that have been hidden by a derived class.

Set attribute In general, JPype only allows the setting of public non-final fields. If you attempt to set any attribute on an object that does not correspond to a settable field it will produce an `AttributeError`. There is one exception to this rule. Sometime it is necessary to attach addition private meta data to classes and objects. Attributes that begin with an underbar are consider to be Python private attributes. Private attributes handled by the default Python attribute handler allowing these attributes to be attached to to attach data to the Python handle. This data is invisible to Java and it is retained only on the Python instance. If an object with Python meta data is passed to Java and Java returns the object, the new Python handle will not contain any of the attached data as this data was lost when the object was passed to Java.

`class_` Attribute For Java classes there is a special attribute called `class`. This is a keyword in Python so *name mangling* applies. This is a class instance of type `java.lang.Class`. It can be used to access fields and methods.

Inner classes For methods and fields, public inner classes appear as attributes of the class. These are regular types that can be used to construct objects, create array, or cast.

String The Java method `toString` is mapped into the Python function `str(obj)`.

Equality The Java method `equals()` has been mapped to Python == with special augmentations for null pointers. Java == is not exposed directly as it would lead to numerous errors. In principle, Java == should map to the Python concept of `is` but it is not currently possible to overload Python in such a way to achieve the desired effect.

Hash The Java method `hashCode` is mapped to Python `hash(obj)` function. There are special augmentations for strings and nulls. Strings will return the same hash code as returned by Python so that Java strings and Python strings produce the same dictionary lookups. Null pointers produce the same hash value as None.

Java defines `hashCode` on many objects including mutable ones. Often the `hashCode` for a mutable object changes when the object is changed. Only use immutable Java object (String, Instant, Boxed types) as dictionary keys or risk undefined behavior.

Java objects are instances of Java classes and have all of the methods defined in the Java class including static members. However, the get attribute method converts public instance members and fields into descriptors which act on the object.

Now that we have defined the basics of Java objects and classes, we will define a few special classes that operate a bit differently.

Array Classes

In Java all arrays are also objects, but they cannot define any methods beyond a limited set of Java array operations. These operations have been mapped into Python to their closest Python equivalent.

Arrays also have a special type factory to produce them. In principle one can create an array class using `JClass` but the signature required would need to use the proper name as required for the Java method `java.lang.Class.forName`. Instead we call the factory to create a new type to use.

The signature for JArray is `JArray(type, [dims=1])`. The type argument accepts any Java type including primitives and constructs a new array class. This class can be used to create new instances, cast, or as the input to the array factory. The resulting object has a constructor method which take either a number, which is the desired size of the array, or a sequence which hold the elements of the array. If the members of the initializer sequence are not Java members then each will be converted. If any element cannot be converted a `TypeError` will be raised.

JArray is an abstract base class for all Java classes that are produced. Thus, one can test if something is an array class using `issubclass` and if Java object is an array using `isinstance`.

Java arrays provide a few additional Python methods:

Get Item Arrays are of course a collection of elements. As such array elements can be accessed using the Python `[]` operator. For multidimensional arrays JPype uses Java style access with a series of index operations such as `jarray[4][2]` rather than NumPy like multidimensional access.

Get Slice Arrays can be accessed using a slice like a Python list. The slice operator is `[start:stop:step]`. It should be noted that array slice are in fact views to the original array so any alteration to the slice will affect the original array. Array slices are cloned when passed back to Java. To force a clone immediately, use the `clone` method. Please note that applying the slice operator to a slice produces a new slice. Thus there can sometimes be an ambiguity between multidimensional access and repeated slicing.

Set Item Array items can be set using the Python `[]=` operator.

Set Slice Multiple array items can be set using a slice assigned with a sequence. The sequence must have the same length as the slice. If this condition is not met, an exception will be raised. If the items to be transferred are a buffer, then a faster buffer transfer assignment will be used. When buffer transfers are used individual elements are not checked for range, but instead cast just like NumPy. Thus, if we have the elements we wish to assign to the array contained within a NumPy array named `na` we can transfer all of them using `jarray[:] = na`.

Buffer transfer Buffer transfers from a Java array also work for primitive types. Thus we can simply call the Python `memoryview(jarray)` function to create a buffer that can be used to transfer any portion of a Java array out. Memory views of Java arrays are not writable.

For each Java arrays can be used as the input to a Python for statement. To iterate each element use `for elem in jarray:`. They can also be used in list comprehensions.

Clone Java arrays can be duplicated using the method clone. To create a copy call `jarray.clone()`. This operates both on arrays and slice views.

Length Arrays in Java have a defined an immutable length. As such the Python `len(array)` function will produce the array length. However, as that does not match Java expectations, JPype also adds an attribute for length so that Java idiom `jarray.length` also works as expected.

In addition, the Java class `JChar[]` has some addition customizations to help work better with string types.

Java arrays are currently missing some of the requirements to act as a `collections.abc.Sequence`. When working with Java arrays it is also useful to use the Java array utilities class `java.util.Arrays` as it has many methods that provide additional functionality. Java arrays do not support any additional mathematical operations at this time.

Buffer classes

In addition to array types, JPype also supports Java `nio` buffer types. Buffers in Java come in two flavors. Array backed buffers have no special access. Direct buffers are can converted to Python buffers with both read and write capabilities.

Each primitive type in Java has its own buffer type named based on the primitive type. `java.nio.ByteBuffer` has the greatest control allowing any type to be read and written to it. Buffers in Java function are like memory mapped files and have a concept of a read and write pointer which is used to traverse the array. They also have direct index access to their specified primitive type.

Java buffer provide an additional Python method:

Buffer transfer Buffer transfers from a Java buffer works for a direct buffer. Array backed buffers will raise a `BufferError`. Use the Python `memoryview(jarray)` function to create a buffer that can be used to transfer any portion of a Java buffer out. Memory views of Java buffers are readable and writable.

Buffers do not currently support element-wise access.

Boxed Classes

Often one wants to be able to place a Java primitive into a method of fields that only takes an object. The process of creating an object from a primitive is referred to as creating a "boxed" object. The resulting object is an immutable object which stores just that one primitive.

Java boxed types in JPype are wrapped with classes that inherit from Python `int` and `float` types as both are immutable in Python. This means that a boxed type regardless of whether produced as a return or created explicitly are treated as Python types. They will obey all the conversion rules corresponding to a Python type as implicit matches.

In addition, they produce an exact match with their corresponding Java type. The type conversion for this is somewhat looser than Java. While Java provides automatic unboxing of a Integer to a double primitive, JPype can implicitly convert Integer to a Double boxed.

To box a primitive into a specific type such as to place it into a `java.util.List` use `JObject` on the desired boxed type or call the constructor for the desired boxed type directly. For example:

```
lst = java.util.ArrayList()
lst.add(JObject(JInt(1)))       # Create a Java integer and box it
lst.add(java.lang.Integer(1))   # Explicitly create the desired boxed object
```

JPype boxed classes have some additional functionality. As they inherit from a mathematical type in Python they can be used in mathematical operations. But unlike Python numerical types they can take an addition state corresponding to being equal to a null pointer. The Python methods are not aware of this new state and will treat the boxed type as a zero if the value is a null.

To test for null, cast the boxed type to a Python type explicitly and the result will be checked. Casting null pointer will raise a `TypeError`.

```
b = JObject(None, java.lang.Integer)
a = b+0      # This succeeds and a gets the value of zero
a = int(b)+0 # This fails and raises a TypeError
```

Boxed objects have the following additional functionality over a normal object.

Convert to index Integer boxed types can be used as Python indices for arrays and other indexing tasks. This method checks that the value of the boxed type is not null.

Convert to int Integer and floating point boxed types can be cast into a Python integer using the `int()` method. The resulting object is always of type `int`. Casting a null pointer will raise a `TypeError`.

Convert to float Integer and floating point boxed types can be cast into a Python float using the `float()` method. The resulting object is always of type `float`. Casting a null pointer will raise a `TypeError`.

Comparison Integer and floating point types implement the Python rich comparison API. Comparisons for null pointers only succeed for == and != operations. Non-null boxed types act like ordinary numbers for the purposes of comparison.

Number Class

The Java class `java.lang.Number` is a special type in Java. All numerical Java primitives and Python number types can convert implicitly into a Java Number.

Input	Result
None	java.lang.Number(null)
Python int, float	java.lang.Number
Java byte, NumPy int8	java.lang.Byte
Java short, NumPy int16	java.lang.Short
Java int, NumPy int32	java.lang.Integer
Java long, NumPy int64	java.lang.Long
Java float, NumPy float32	java.lang.Float
Java double, NumPy float64	java.lang.Double

Additional user defined conversion are also applied. The primitive types boolean and char and their corresponding boxed types are not considered to numbers in Java.

Object Class

Although all classes inherit from Object, the object class itself has special properties that are not inherited. All Java primitives will implicitly convert to their box type when placed in an Object. In addition, a number of Python types implicitly convert to a Java object. To convert to a different object type, explicitly cast the Python object prior to placing in a Java object.

Here a table of the conversions:

Input	Result
None	java.lang.Object(null)
Python str	java.lang.String
Python bool	java.lang.Boolean
Python int	java.lang.Number
Python float	java.lang.Number

In addition it inherits the conversions from `java.lang.Number`. Additional user defined conversion are also applied.

String Class

The String class in Java is a special representation often pointing either to a dynamically created string or to a constant pool item defined in the class. All Java strings are immutable just like Python strings and thus these are considered to be equivalent classes.

Because Java strings are in fact just pointers to blob of bytes they are actually slightly less than a full object in some JVM implementation. This is a violation of the Object Orients (OO) principle, never take something away by inheritance. Unfortunately, Java is a frequent violator of that rule, so this is just one of those exceptions you have to trip over. Therefore, certain operations such as using a string as a threading control with `notify` or `wait` may lead to unexpected results. If you are thinking about using a Java string in synchronized statement then remember it is not a real object.

Java strings have a number of additional functions beyond a normal object.

Length Java strings have a length measured in the number of characters required to represent the string. Extended Unicode characters count for double for the purpose of counting characters. The string length can be determined using the Python `len(str)` function.

Indexing Java strings can be used as a sequence of characters in Python and thus each character can be accessed as using the Python indexing operator `[]`.

Hash Java strings use a special hash function which matches the Python hash code. This ensures that they will always match the same dictionary keys as the corresponding string in Python. The Python hash can be determined using the Python `hash(str)` function. Null pointers are not currently handled. To get the actually Java hash, use `s.hashCode()`

Contains Java strings implement the concept of `in` when using the Java method `contains`. The Java implementation is sufficiently similar that it will work fairly well on strings. For example, `"I" in java.lang.String("team")` would be equal to False.

Testing other types using the `in` operator will likely raise a `TypeError` if Java is unable to convert the other item into something that can be compared with a string.

Concatenation Java strings can be appended to create a new string which contains the concatenation of the two strings. This is mapped to the Python operator +.

Comparison Java strings are compared using the Java method `compareTo`. This method does not currently handle null and will raise an exception.

For each Java strings are treated as sequences of characters and can be used with a for-loop construct and with list comprehension. To iterate through all of the characters, use the Python construct `for c in str:`.

Unfortunately, Java strings do not yet implement the complete list of requirements to act as Python sequences for the purposes of `collections.abc.Sequence`.

The somewhat outdated JString factory is a Python class that pretends to be a Java string type. It has the marginal advantage that it can be imported before the JVM is actually started. Once the JVM is started, its class representation is pointed to `java.lang.String` and can be used to construct a new string object or to test if an object is actually a Java string using `isinstance`. It does not implement any of the other string methods and just serves as convenience class. The more capable `java.lang.String` can be imported in place of JString, but only after the JVM is started.

String objects may optionally convert to Python strings when returned from Java methods, though this option is a performance issue and can lead to other difficulties. This setting is selected when the JVM is started. See *String Conversions* for details.

Java strings will cache the Python conversion so we only pay the conversion cost once per string.

Exception Classes

Both Python and Java treat exception classes differently from other objects. Only these types may be caught as part of a try block. Therefore, the exceptions have a special wrapper. Most of the mechanics of exceptions happen under the surface. The one difference between Python and Java is the behavior when the argument is queried. Java arguments can either be the string value, the exception itself, or the internal construction key depending on how the exception came into existence. Therefore, the arguments to a Java exception should never be used as their values are not guaranteed.

Java exception can report their stacktrace to Python in two different ways. If printed through the Python stack trace routine, Java exceptions are split between the Python code that raised and a phantom Java `cause` which contains the Java exception in Python order. If the debugging information for the Java source is enabled, Python may even print the Java source code lines where the error occurred. If you prefer Java style stack traces then print the result from the `stacktrace()` method. Unhandled exception that terminate the program will print the Python style stack trace information.

The base class `JException` is a special type located in `jpype.types` that can be imported prior to the start of the JVM. This serves as the equivalent of `java.lang.Throwable` and contains no additional methods. It is currently being phased out in favor of catching the Java type directly.

Using `jpype.JException` with a class name as a string was supported in previous JPype versions but is currently deprecated. For further information on dealing with exception, see the *Exception Handling* section. To create a Java exception use JClass or any of the other importing methods.

Anonymous Classes

Sometimes Java will produce an anonymous class which does to have any actual class representation. These classes are generated when a method implements a class directly as part of its body and they serve as a closure with access to some of the variables that were used to create it.

For the purpose of JPype these classes are treated as their parents. But this is somewhat problematic when the parent is simply an interface and not an actual object type.

Lambdas

The companion of anonymous classes are lambda classes. These are generated dynamically and their parent is always an interface. Lambdas are always Single Abstract Method (SAM) type interfaces. They can implement additional methods in the form of default methods but those are generally not accessible within JPype.

Inner Classes

For the most part, inner classes can be used like normal classes, with the following differences:

- Inner classes in Java natively use $ to separate the outer class from the inner class. For example, inner class Foo defined inside class Bar is called Bar.Foo in Java, but its real native name is Bar$Foo.
- Inner classes appear as member of the containing class. Thus to access them import the outer class and call them as members.
- Non-static inner classes cannot be instantiated from Python code. Instances received from Java code can be used without problem.

Importing Java classes

As Java classes are remote from Python and can neither be created nor extended within Python, they must be imported. JPype provides three different methods for creating classes.

The highest level API is the use of the import system. To import a Java class, one must first import the optional module `jpype.imports` which has the effect of binding the Java package system to the Python module lookup. Once this is completed package or class can be imported using the standard Python import system. The import system offers a very rich error reporting system. All failed imports produce an `ImportError` with diagnostics as to what went wrong. Errors include unable to find the class, unable to find a required dependency, and incorrect Java version.

One important caveat when dealing with importing Java modules. Python always imports local directories as modules before calling the Java importer. So any directory named `java`, `com`, or `org` will hide corresponding Java package. We recommend against naming directories as `java` or top level domain.

The older method of importing a class is with the `JPackage` factory. This factory automatically loads classes as attributes as requested. If a class cannot be found it will produce an `AttributeError`. The symbols `java` and `javax` in the `jpype` module are both `JPackage` instances. Only public classes appear on `JPackage` but protected and even private classes can be accessed by name. Though most private classes don't have any methods or fields that can be accessed.

The last mechanism for looking up a class is through the use of the `JClass` factory. This is a low level API allowing the loading of any class available using the forName mechanism in Java. The JClass method can take up to three arguments corresponding to arguments of the forName method and can be used with alternative class loaders. The majority of the JPype test bench uses JClass so that the tests are only evaluating the desired functionality and not the import system. But this does not imply that JClass is the preferred mechanic for importing classes. The first argument can be a string or a Java class instance. There are two keyword arguments `loader` and `initialize`. The loader can point to an alternative ClassLoader which is handy when loading custom classes through mechanisms such as over the web. A False `initialize` argument loads a class without loading dependencies nor populating static fields. This option is likely not useful for ordinary users. It was provided when calling forName was problematic due to *caller sensitive* issues.

Type Matching

This section provides tables documenting the JPype conversion rules. JPype defines different levels of "match" between Python objects and Java types. These levels are:

- **none**, There is no way to convert.
- **explicit (E)**, JPype can convert the desired type, but only explicitly via casting. Explicit conversions are only execute automatically in the case of a return from a proxy.
- **implicit (I)**, JPype will convert as needed.
- **exact (X)**, Like implicit, but when deciding with method overload to use, one where all the parameters match "exact" will take precedence over "implicit" matches.

See the previous section on *Java Conversions* for details.

There are special conversion rules for `java.lang.Object` and `java.lang.Number`. (*Object Class* and *Number Class*)

Python\Java	byte	short	int	long	float	double	boolean	char	String	Array	Object	java.lang.Object	java.lang.Class
int	I[1]	I[1]	X	I	I[3]	I[3]	X[8]					I[11]	
long	I[1]	I[1]	I[1]	X	I[3]	I[3]						I[11]	
float					I[1]	X						I[11]	
sequence													
dictionary													
string								I[2]	X			I	
unicode								I[2]	X			I	
JByte	X											I[9]	
JShort		X										I[9]	
JInt			X									I[9]	
JLong				X								I[9]	
JFloat					X							I[9]	
JDouble						X						I[9]	
JBoolean							X					I[9]	
JChar								X				I[9]	
JString									X			I	
JArray										I/X[4]		I	
JObject										I/X[6]	I/X[7]	I/X[7]	
JClass												I	X
"Boxed"[10]	I	I	I	I	I	I	I					I	

Exception Handling

Error handling is an important part of any non-trivial program. All Java exceptions occurring within Java code raise a `jpype.JException` which derives from Python Exception. These can be caught either using a specific Java exception or generically as a `jpype.JException` or `java.lang.Throwable`. You can then use the `stacktrace()`, `str()`, and args to access extended information.

Here is an example:

```
try :
      # Code that throws a java.lang.RuntimeException
except java.lang.RuntimeException as ex:
      print("Caught the runtime exception : ", str(ex))
      print(ex.stacktrace())
```

Multiple java exceptions can be caught together or separately:

[1] Conversion will occur if the Python value fits in the Java native type.

[3] Java defines conversions from integer types to floating point types as implicit conversion. Java's conversion rules are based on the range and can be lossy. See (http://stackoverflow.com/questions/11908429/java-allows-implicit-conversion-of-int-to-float-why)

[8] Only the values *True* and *False* are implicitly converted to booleans.

[11] Boxed to `java.lang.Number`

[2] Conversion occurs if the Python string or unicode is of length 1.

[9] Primitives are boxed as per Java rules.

[4] Number of dimensions must match and the types must be compatible.

[6] Only if the specified type is a compatible array class.

[7] The object class is an exact match, otherwise implicit.

[10] Java boxed types are mapped to Python primitives, but will produce an implicit conversion even if the Python type is an exact match. This is to allow for resolution between methods that take both a Java primitve and a Java boxed type.

```
try:
      # ...
except (java.lang.ClassCastException, java.lang.NullPointerException) as ex:
      print("Caught multiple exceptions : ", str(ex))
      print(ex.stacktrace())
except java.lang.RuntimeException as ex:
      print("Caught runtime exception : ", str(ex))
      print(ex.stacktrace())
except jpype.JException:
      print("Caught base exception : ", str(ex))
      print(ex.stacktrace())
except Exception as ex:
      print("Caught python exception :", str(ex))
```

Exceptions can be raised in proxies to throw an exception back to Java.

Exceptions within the JPype core are issued with the most appropriate Python exception type such as `TypeError`, `ValueError`, `AttributeError`, or `OSError`.

Exception aliasing

Certain exceptions in Java have a direct correspondence with existing Python exceptions. Rather than forcing JPype to translate these exceptions, or forcing the user to handle Java exception types throughout the code, we have "derived" these exceptions from their Python counter parts. Thus, rather than requiring special error handling for Java you can simple catch these exceptions using the standard Python exception types.

java.lang.IndexOutOfBoundsException This exception is synonymous with the Python exception `IndexError`. As many slicing or array operations in Java can produce an IndexOutOfBoundsException but the Python contract for slicing of an array should raise an `IndexError`, this type has been customized to consider IndexError to be a base type.

java.lang.NullPointerException This exception is derived from the Python exception `ValueError`. Numerous Java calls produce a `NullPointerException` and in all cases this would match a Python `ValueError`.

By deriving these exceptions from Python, the user is free to catch the exception either as a Java exception or as the more general Python exception. Remember that Python exceptions are evaluated in order from most specific to least.

1.2.4 Controlling the JVM

In this chapter, we will discuss how to control the JVM from within Python. For the most part, the JVM is invisible to Python. The only user controls needed are to start up and shutdown the JVM.

Starting the JVM

The first task is always to start the JVM. The settings to the JVM are immutable over the lifespan of the JVM. The user settings are: the JVM arguments, the class path used to find jars, and whether to convert Java strings to Python strings.

Class paths

JPype supports two styles of classpaths. The first is modeled after Matlab the second argument style uses a list to the `startJVM` function.

The Matlab style uses the functions `jpype.addClassPath` and `getClassPath`. The first function adds a directory or jar file to the search path. Wild cards are accepted in the search. Once all of the paths are added to internal class path, they can be retrieved using `getClassPath` which takes a keyword argument `env` which defaults to true. When set to false, JPype will ignore the environment variable `CLASSPATH` which is normally included in the default classpath.

To use the argument style, pass all of the class paths in a list as the keyword argument `classpath` to the `startJVM`. This classpath method does not include the environment `CLASSPATH`, but it does provide a quick method to pull in a specific set of classes. Wild cards are accepted as the end of the path to include all jars in a given directory.

One should note that the class path can only be set prior starting the JVM. Calls to set the class path after the JVM is started are silently ignored. If a jar must be loaded after the JVM is started, it may be loaded using `java.net.URLClassLoader`. Classes loaded using a `URLClassloader` are not visible to JPype imports nor to JPackage.

String conversions

The `convertStrings` argument defines how strings are returned by JPype. Early in the life of this project return types were often converted to Python types without regard to preserving the type information. Thus strings would automatically convert to a Python string effectively the data from Java to Python on each return. This was a violation of the Python philosophy that explicit is better than implicit. This also prohibited chaining of Java string operations as each operation would lose the Java representation and have to be transferred back and forth. The simple operation of trying to create a Java string was difficult as directly calling `java.lang.String` constructor would once again convert the result back to a Python string, hence the need to use the `JString` factory. There was an option to turn off the conversion of strings, but it was never operable. Therefore, all code written at the time would expect Java strings to convert to Python strings on return.

Recognizing this is both a performance issue and that it made certain types of programming prohibitive, JPype switched to having a setting requiring applications to chose a policy at the start of operation. This option is a keyword argument `convertStrings`. The default for 0.7 is to give the older broken behavior. If specified as False, Java strings will act as ordinary classes and return a Java string instance. This string instance can be converted by calling the Python `str()` function. Failure to specify a policy will issue a warning message.

You are strongly encouraged to set convertStrings false especially when are writing reusable Python modules with JPype. String in JPype 0.8, the default will to not convert strings.

Path to the JVM

In order the start the JVM, JPype requires the path to the Java shared library typically located in the JRE installation. This can either be specified manually as the first argument to `jpype.startJVM` or by automatic search.

The automatic search routine uses different mechanisms depending on the platform. Typically the first mechanism is the use the environment variable `JAVA_HOME`. If no suitable JVM is found there, it will then search common directories based on the platform. On windows it will consult the registry.

You can get the JVM found during the automatic search by calling `jpype.getDefaultJVMPath()`.

In order to use the JVM, the architecture of the JVM must match the Python version. A 64 bit Python can only use a 64 bit JVM. If no suitable JVM can be found it should raise an error. In some cases so rare, it may lead to a crash depending on how the platform handles a failed shared library load.

Launching the JVM

Now that we have discussed the JVM options, lets show how to put it into practice. Suppose that the Python script at the top level of your working director, with a subdirectory holding all your working jars `./lib`, and a second

directory with bare classes `./classes`. Java has been properly installed with the same architecture as Python (both 64 bit in this case).

To start JPype we would execute the following:

```
import jpype
jpype.startJVM("-ea", classpath=['lib/*', 'classes'], convertStrings=False)
```

Arguments that begin with a dash are passed to the JVM. Any unrecognized argument will raise an exception unless the keyword argument `ignoreUnrecognized` is set to `True`. Details of available arguments can be found in the vendor JVM documentation.

The most frequent problem encountered when starting JPype is the jars failing to be loaded. Java is unforgiving when loading jar files. To debug the failures, we will need to print the loaded classpath.

Java has a method to retrieve the classpath that was used during the loading process.

```
print(java.lang.System.getProperty('java.class.path'))
```

This command will print the absolute path to each of the jars that will be used by the JVM. Each of the jars are written out explicitly as the JVM does not permit wild-cards. JPype has expanded each of them using *glob*. If an expected jar file is missing the list, then it will not be accessable.

There is a flag to determine the current state of the JVM. Calling `jpype.isJVMStarted()` will return the current state of the JVM.

Once the JVM is started, we can find out the version of the JVM. The JVM can only load jars and classfiles compiled for the JVM version or older. Newer jar files will invariably fail to load. The JVM version can be determined using `jpype.getJVMVersion()`.

Shutting down the JVM

At the other end of the process after all work has been performed, we will want to shutdown the JVM to terminate the program. This will happen automatically and no user intervention is required. If however, the user wants to continue execution of Python code after the JVM is finished they can explicitly call `jpype.shutdownJVM()`. This can only be called from the main Python thread. Any other thread will raise an exception.

The shutdown procedure of JPype and Java is fairly complicated.

1) JPype requests that the JVM shutdown gracefully.
2) Java waits until all non-daemon thread terminate. Thus if you did not send a termination to each non-daemon threads the shutdown will wait here until those threads complete their work.
3) Once the all threads have completed except for the main thread, the JVM will begin the shutdown sequence. From this point on the JVM is in a crippled state limited what can happen to spawning the shutdown threads and completing them.
4) The shutdown will first spawn the threads of cleanup routine that was attached to the JVM shutdown hook in arbitrary order. These routines can call back to Python and perform additional tasks.
5) Once the last of these threads are completed, JPype then shuts down the reference queue which dereferences held all Python resources.
6) Then JPype shuts down the type manager and frees all internal resources in the JPype module.
7) Last, it unloads the JVM shared library returning the memory used by the JVM.
8) Once that is complete, control is returned to Python.

All Java objects are now considered dead and cannot be reactivated. Any attempt to access their data field will raise an exception.

Attaching a shutdown hook

If you have resources that need to be closed when the JVM is shutdown these should be attached to the Java Runtime object. The following pattern is used:

```
@JImplements(Runnable)
class MyShutdownHook:
    @JOverride
    def run(self):
        # perform any required shutdown activities

java.lang.Runtime.getRuntime().addShutdownHook(Thread(MyShutdownHook()))
```

This thread will be executed in a new thread once the main thread is the only one remaining alive. Care should always be taken to complete work in a timely fashion and be aware the shutdown threads are inherently racing with each other to complete their work. Thus try to avoid expensive operations on shutdown..

Debugging shutdown

The most common failure during shutdown is the failure of an attached thread to terminate. There are specific patterns in Java that allow you to query for all currently attached threads.

1.2.5 Customization

JPype supports three different types of customizations.

The first is to adding a Python base class into a Java tree as was done with certain exceptions. This type of customization required private calls in JPype and is not currently exposed to the user.

Second a Python class can be used as a template when a Java class is first constructed to add additional functionality. This type of customization can be used to make a Java class appear as a native Python class. Many of the Java collection classes have been customized to match Python collections.

Last, Python class can be added to the implicit conversion list. This customizer is used to make Python types compatable with Java without requiring the user to manually case over and over.

All customization available to the users is done through class decorators added to Python classes or functions.

Class Customizers

Java wrappers can be customized to better match the expected behavior in Python. Customizers are defined using decorators. Applying the annotations `@JImplementationFor` and `@JOverride` to a regular Python class will transfer methods and properties to a Java class. `@JImplementationFor` requires the class name as a string, a Java class wrapper, or Java class instance. Only a string can be used prior to starting the JVM. `@JOverride` when applied to a Python method will hide the Java implementationallowing the Python method to replace the Java implementation. when a Java method is overridden, it is renamed with an proceeding underscore to appear as a private method. Optional arguments to `@JOverride` can be used to control the renaming and force the method override to apply to all classes that derive from a base class ("sticky").

Generally speaking, a customizer should be defined before the first instance of a given class is created so that the class wrapper and all instances will have the customization.

Example taken from JPype `java.util.Map` customizer:

```
@_jcustomizer.JImplementationFor('java.util.Map')
class _JMap:
    def __jclass_init__(self):
       Mapping.register(self)

    def __len__(self):
       return self.size()

    def __iter__(self):
       return self.keySet().iterator()

    def __delitem__(self, i):
       return self.remove(i)
```

The name of the class does not matter for the purposes of customizer though it should be a private class so that it does not get used accidentally. The customizer code will steal from the prototype class rather than acting as a base class, thus, ensuring that the methods will appear on the most derived Python class and are not hidden by the java implementations. The customizer will copy methods, callable objects, `__new__`, class member strings, and properties.

Type Conversion Customizers

One can add a custom converter method which is called whenever a specified Python type is passed to a particular Java type. To specify a conversion method add `@JConversion` to an ordinary Python function with the name of Java class to be converted to and one keyword of `exact` or `instanceof`. The keyword controls how strictly the conversion will be applied. `exact` is restricted to Python objects whose type exactly matches the specified type. `instanceof` accepts anything that matches isinstance to the specified type or protocol. In some cases, the existing protocol definition will be overly broad. Adding the keyword argument `excludes` with a type or tuple of types can be used to prevent the conversion from being applied. Exclusions always apply first.

User supplied conversions are tested after all internal conversions have been exhausted and are always considered to be an implicit conversion.

```
@_jcustomizer.JConversion("java.util.Collection", instanceof=Sequence,
  excludes=str)
def _JSequenceConvert(jcls, obj):
    return _jclass.JClass('java.util.Arrays').asList(obj)
```

JPype supplies customizers for certain Python classes.

Python class	Implicit Java Class
pathlib.Path	java.io.File
pathlib.Path	java.nio.file.Path
datetime.datetime	java.time.Instant
collections.abc.Sequence	java.util.Collection
collections.abs.Mapping	java.util.Map

1.2.6 Collections

JPype uses customizers to augment Java collection classes to operate like Python collections. Enhanced objects include `java.util.List`, `java.util.Set`, `java.util.Map`, and `java.util.Iterator`. These classes generally comply with the Python API except in cases where there is a significant name conflict and thus no special

treatment is required when handling these Java types. Details of customizing Java classes can be found in the previous chapter, *Customization*.

This section will detail the various customization that are to applied the Java collection classes.

Iterable

All Java classes that implement `java.util.Iterable` are customized to support Python iterator notation and thus can be used in Python for loops and in list comprehensions.

Iterators

All Java classes that implement `java.util.Iterator` act as Python iterators.

Collection

All Java classes that inherit from `java.util.Collection` have a defined length determined by the Python `len(obj)` function. As they also inherit from Iterable, they have iterator, forech traversal, and list comprehension.

In addition, methods that take a Java collection can convert a Python sequence into a collection implicitly if all of the elements have a conversion into Java. Otherwise a `TypeError` is raised.

Lists

Java List classes such as ArrayList and LinkedList can be used in Python `for` loops and list comprehensions directly. A Java list can be converted to a Python list or the reverse by calling the requested type as a copy constructor.

```
pylist = ['apple', 'orange', 'pears']

# Copy the Python list to Java.
jlist = java.util.ArrayList(pylist)

# Copy the Java list back to Python.
pylist2 = list(jlist)
```

Note that the individual list elements are still Java objects when converted to Python and thus a list comprehension would be required to force Python types if required. Converting to Java will attempt to convert each argument individually to Java. If there is no conversion it will produce a `TypeError`. The conversion can be forced by casting to the appropriate Java type with a list comprehension or by defining a new conversion customizer.

Lists also have iterable, length, item deletion, and indexing. Note that indexing of `java.util.LinkedList` is supported but can have a large performance penalty for large lists. Use of iteration is much for efficient.

Map

A Java classes that implement `java.util.Map` inherit the Python collections.abc.Mapping interface. As such they can be iterated, support the indexing operator for value lookups, item deletion, length, and support contains.

Here is a summary of their capabilities:

Action	Python
Place a value in the map	`jmap[key]=value`
Delete an entry	`del jmap[key]`
Get the length	`len(jmap)`
Lookup the value	`v=jmap[key]`
Get the entries	`jmap.items()`
Fetch the keys	`jmap.key()`
Check for a key	`key in jmap`

In addition, methods that take a Java map can implicitly convert a Python `dict` or a class that implements `collections.abc.Mapping` assuming that all of the map entries can be converted to Java. Otherwise a `TypeError` is raised.

MapEntry

Java map entries unpack into a two value tuple, thus supporting iterating through key value pairs. Thus is useful when iterating map entries in a for loop by pairs.

Set

All Java classes that implement `java.util.Set` implement delitem as well as the Java collection customizations.

Enumeration

All Java classes that implement `java.util.Enumeration` inherit Python iterator behavior and can be used in Python for loops and list comprehensions.

1.2.7 Working with NumPy

As one of the primary focuses of JPype is working with numerical codes such as NumPy, there are a number of NumPy specific enhancements. NumPy is a large binary package and therefore JPype cannot be compiled against NumPy directly without force it to be a requirement. Instead of compiling against NumPy directly, JPype implements interfaces that NumPy can recognize and use. The specific enhancements are the following: direct buffer transfers of primitive arrays and buffers, direct transfer of multi dimensional arrays, buffer backed NumPy arrays, and conversion of NumPy integer types to Java boxed types.

Transfers to Java

Memory from a NumPy array can be transferred Java in bulk. The transfer of a one dimensional NumPy array to Java can either be done at initialization or by use of the Python slice operator.

Assuming we have a single dimensional NumPy array `npa`, we can transfer it with initialization using

```
ja = JArray(JInt)(npa)
```

Or we can transfer it to Java as a slice assignment.

```
ja[:] = npa
```

The slice operator can transfer the entire array or just a portion of it.

Multidimensional transfers to Java

Multidimensional arrays can also be transferred at initialization time. To transfer a NumPy array to Java use the `JArray.of` function

```
z = np.zeros((5,10,20))
ja = JArray.of(z)
```

Transfers to NumPy

Java arrays can be in two forms. Java multidimensional arrays are not contiguous in memory. If all of the arrays in each dimension are the same, then the array is rectangular. If the size of the arrays within any dimension differ, then the array is jagged. Jagged arrays are an array of arrays rather than a rectangular block of memory.

NumPy arrays only hold rectangular arrays as multidimensional arrays of primitives. All other arrangements are a stored as a single dimensional array of objects. JPype can automatically transfer a rectangular array to NumPy as a bulk transfer. To do so JPype supports a `memoryview` on rectangular arrays. Whenever a memoryview is called on a multidimensional array of primitives, JPype verifies that it is rectangular and creates a buffer. If it is jagged, a `BufferError` is raised. When a Java array is used as an argument to initialize a NumPy array, it creates a `memoryview` so that all of the memory can be transferred in bulk.

Buffer backed NumPy arrays

Java direct buffers provide access between foreign memory and Java. This access bypasses the JNI layer entirely, permitting Java and Python to operate on a memory space with native speed. Java direct buffers are not under the control of the garbage collector and thus can result in memory leaks and memory exhaustion if not used carefully. This is used with Java libraries that support direct buffers. Direct buffers are part of the Java `nio` package and thus functionality for buffers is in `jpype.nio`.

To create a buffer backed NumPy array, the user must either create a direct memory buffer using the Java direct buffer API or create a Python `bytearray` and apply `jpype.nio.convertToByteBuffer` to map this memory into Java space. NumPy can then convert the direct buffer into an array using `asarray`.

To originate a direct buffer from Java, use:

```
jb = java.nio.ByteBuffer.allocateDirect(80)
db = jb.asDoubleBuffer()
a = np.asarray(db)
```

To origate a direct buffer from Python, use:

```
bb = bytearray(80)
jb = jpype.nio.convertToDirectBuffer(bb)
db = jb.asDoubleBuffer()
a = np.asarray(db)
```

Buffer backed arrays have one downside. Python and by extension NumPy have no way to tell when a buffer becomes invalid. Once the JVM is shutdown, all buffers become invalid and any access to NumPy arrays backed by Java risk crashing. To avoid this fate, either create the memory for the buffer from within Python and pass it to Java. Or use the Java `java.lang.Runtime.exit` which will terminate both the Java and Python process without leaving any opertunity to access a dangling buffer.

Buffer backed memory is not limited to use with NumPy. Buffer transfers are supported to provide shared memory between processes or memory mapped files. Anything that can be mapped to an address with as a flat array of primitives with machine native byte ordering can be mapped into Java.

NumPy Primitives

When converting a Python type to a boxed Java type, there is the difficulty that Java has no way to known the size of a Python numerical value. But when converting NumPy numerical types, this is not an issue. The following conversions apply to NumPy primitive types.

Numpy Type	Java Boxed Type
np.int8	java.lang.Byte
np.int16	java.lang.Short
np.int32	java.lang.Integer
np.int64	java.lang.Long
np.float32	java.lang.Float
np.float64	java.lang.Double

Further, these NumPy types obey Java type conversion rules so that they act as the equivalent of the Java primitive type.

1.2.8 Implementing Java interfaces

Proxies in Java are foreign elements that pretend to implement a Java interface. We use this proxy API to allow Python to implement any Java interface. Of course, a proxy is not the same as subclassing Java classes in Python. However, most Java APIs are built so that sub-classing is not required. Good examples of this are AWT and SWING. Except for relatively advanced features, it is possible to build complete UIs without creating a single subclass.

For those cases where sub-classing is absolutely necessary (i.e. using Java's SAXP classes), it is necessaryy to create an interface and a simple subclass in Java that delegates the calls to that interface. The interface can then be implemented in Python using a proxy.

There are two APIs for supporting of Java proxies. The new high-level interface uses decorators which features strong error checking and easy notation. The older low-level interface allows any Python object or dictionary to act as a proxy even if it does not provide the required methods for the interface.

Implements

The newer style of proxy works by decorating any ordinary Python class to designate it as a proxy. This is most effective when you control the Python class definition. If you don't control the class definition you either need to encapsulate the Python object in another object or use the older style.

Implementing a proxy is simple. First construct an ordinary Python class with method names that match the Java interface to be implemented. Then add the `@JImplements` decorator to the class definition. The first argument to the decorator is the interface to implement. Then mark each method corresponding to a Java method in the interface with `@JOverride`. When the proxy class is declared, the methods will be checked against the Java interface. Any missing method will result in JPype raising an exception.

High-level proxies have one other important behavior. When a proxy created using the high-level API returns from Java it unpacks back to the original Python object complete with all of its attributes. This occurs whether the proxy is the `self` argument for a method or proxy is returned from a Java container such as a list. This is accomplished because the actually proxy is a temporary Java object with no substance, thus rather than returning a useless object, JPype unpacks the proxy to its original Python object.

Proxy Method Overloading

Overloaded methods will issue to a single method with the matching name. If they take different numbers of arguments then it is best to implement a method dispatch:

```
@JImplements(JavaInterface)
class MyImpl:
   @JOverride
   def callOverloaded(self, *args):
     # always use the wild card args when implementing a dispatch
     if len(args)==2:
        return self.callMethod1(*args)
     if len(args)==1 and isinstance(args[0], JString):
        return self.callMethod2(*args)
     raise RuntimeError("Incorrect arguments")

   def callMethod1(self, a1, a2):
     # ...
   def callMethod2(self, jstr):
     # ...
```

Multiple interfaces

Proxies can implement multiple interfaces as long as none of those interfaces have conflicting methods. To implement more than one interface, use a list as the argument to the JImplements decorator. Each interface must be implemented completely.

Deferred realization

Sometimes it is useful to implement proxies before the JVM is started. To achieve this, specify the interface using a string and add the keyword argument `deferred` with a value of `True` to the decorator.

```
@JImplements("org.foo.JavaInterface", deferred=True)
class MyImpl:
   # ...
```

Deferred proxies are not checked at declaration time, but instead at the time for the first usage. Because of this, when uses an deferred proxy the code must be able to handle initialization errors wherever the proxy is created.

Other than the raising of exceptions on creation, there is no penalty to deferring a proxy class. The implementation is checked once on the first usage and cached for the remaining life of the class.

Proxy Factory

When a foreign object from another module for which you do not control the class implementation needs to be passed into Java, the low level API is appropriate. In this API you manually create a JProxy object. The proxy object must either be a Python object instance or a Python dictionary. Low-level proxies use the JProxy API.

JProxy

The `JProxy` allows Python code to "implement" any number of Java interfaces, so as to receive callbacks through them. The JProxy factory has the signature:

```
JProxy(intr, [dict=obj | inst=obj] [, deferred=False])
```

The first argument is the interface to be implemented. This may be either a string with the name of the interface, a Java class, or a Java class instance. If multiple interfaces are to be implemented the first argument is replaced by a Python sequence. The next argument is a keyword argument specifying the object to receive methods. This can either be a dictionary `dict` which names the methods as keys or an object instance `inst` which will receive method calls. If more than one option is selected, a `TypeError` is raised. When Java calls the proxy the method is looked up in either the dictionary or the instance and the resulting method is called. Any exceptions generated in the proxy will be wrapped as a `RuntimeException` in Java. If that exception reaches back to Python it is unpacked to return the original Python exception.

Assume a Java interface like:

```
public interface ITestInterface2
{
        int testMethod();
        String testMethod2();
}
```

You can create a proxy *implementing* this interface in two ways. First, with an object:

```
class C :
        def testMethod(self) :
                return 42

        def testMethod2(self) :
                return "Bar"

c = C()  # create an instance
proxy = JProxy("ITestInterface2", inst=c) # Convert it into a proxy
```

or you can use a dictionary.

```
def _testMethod() :
   return 32

def _testMethod2() :
   return "Fooo!"

d = { 'testMethod' : _testMethod, 'testMethod2' : _testMethod2, }
proxy = JProxy("ITestInterface2", dict=d)
```

Proxying Python objects

Sometimes it is necessary to push a Python object into Java memory space as an opaque object. This can be achieved using be implementing a proxy for an interface which has no methods. For example, `java.io.Serializable` has no arguments and little functionality beyond declaring that an object can be serialized. As low-level proxies to not automatically convert back to Python upon returning to Java, the special keyword argument `convert` should be set to True.

For example, let's place a generic Python object such as NumPy array into Java.

```
import numpy as np
u = np.array([[1,2],[3,4]])
ls = java.util.ArrayList()
```

(continues on next page)

(continued from previous page)

```
ls.add(jpype.JProxy(java.io.Serializable, inst=u, convert=True))
u2 = ls.get(0)
print(u is u2)  # True!
```

We get the expected result of `True`. The Python has passed through Java unharmed. In future versions of JPype, this method will be extended to provide access to Python methods from within Java by implementing a Java interface that points to back to Python objects.

Reference Loops

It is strongly recommended that object used in proxies must never hold a reference to a Java container. If a Java container is asked to hold a Python object and the Python object holds a reference to the container, then a reference loop is formed. Both the Python and Java garbage collectors are aware of reference loops within themselves and have appropriate handling for them. But the memory space of the other machine is opaque and neither Java nor Python is aware of the reference loop. Therefore, unless you manually break the loop by either clearing the container, or removing the Java reference from Python these objects can never be collected. Once you lose the handle they will both become immortal.

Ordinarily the proxy by itself would form a reference loop. The Python object points to a Java invocation handler and the invocation handler points back to Python object to prevent the Python object from going away as long as Java is holding onto the proxy. This is resolved internally by making the Python weak reference the Java portion. If Java ever garbage collects the Java half, it is recreated again when the proxy is next used.

This does have some consequences for the use of proxies. Proxies must never be used as synchronization objects. Whenever they are garbage collected, they loss their identity. In addition, their hashCode and system id both are reissued whenever they are refreshed. Therefore, using a proxy as a Java map key can be problematic. So long as it remains in the Java map, it will maintain the same identify. But once it is removed, it is free to switch identities every time it is garbage collected.

1.2.9 Concurrent Processing

This chapter covers the topic of threading, synchronization, and multiprocess. Much of this material depends on the use of *Proxies* covered in the prior chapter.

Threading

JPype supports all types of threading subject to the restrictions placed by Python. Java is inherently threaded and support a vast number of threading styles such as execution pools, futures, and ordinary thread. Python is somewhat more limited. At its heart Python is inherently single threaded and requires a master lock known as the GIL (Global Interpreter Lock) to be held every time a Python call is made. Python threads are thus more cooperative that Java threads.

To deal with this behavior, JPype releases the GIL every time it leaves from Python into Java to any user defined method. Shorter defined calls such as to get a string name from from a class may not release the GIL. Every time the GIL is released it is another opportunity for Python to switch to a different cooperative thread.

Python Threads

For the most part, Python threads based on OS level threads (i.e. POSIX threads) will work without problem. The only challenge is how Java sees threads. In order to operate on a Java method, the calling thread must be attached to Java. Failure to attach a thread will result in a segmentation fault. It used to be a requirement that users manually attach their

thread to call a Java function, but as the user has no control over the spawning of threads by other applications such as an IDE, this inevitably lead to unexpected segmentation faults. Rather that crashing randomly, JPype automatically attachs any thread that invokes a Java method. These threads are attached automatically as daemon threads so that will not prevent the JVM from shutting down properly upon request. If a thread must be attached as a non-daemon, use the method `jpype.attachThreadToJVM()` from within the thread context. Once this is done the JVM will not shut down until that thread is completed.

There is a function called `jpype.isThreadAttachedToJVM()` which will check if a thread is attached. As threads automatically attach to Java, the only way that a thread would not be attached is if it has never called a Java method.

The downside of automatic attachment is that each attachment allocates a small amount of resources in the JVM. For applications that spawn frequent dynamically allocated threads, these threads will need to be detached prior to completing the thread with `jpype.detachThreadFromJVM()`. When implementing dynamic threading, one can detach the thread whenever Java is no longer needed. The thread will automatically reattach if Java is needed again. There is a performance penalty each time a thread is attached and detached.

Java Threads

To use Java threads, create a Java proxy implementins `java.lang.Runnable`. The Runnable can then be passed any Java threading mechanism to be executed. Each time that Java threads transfer control back to Python, the GIL is reacquired.

Other Threads

Some Python libraries offer other kinds of thread, (i.e. microthreads). How they interact with Java depends on their nature. As stated earlier, any OS- level threads will work without problem. Emulated threads, like microthreads, will appear as a single thread to Java, so special care will have to be taken for synchronization.

Synchronization

Java synchronization support can be split into two categories. The first is the `synchronized` keyword, both as prefix on a method and as a block inside a method. The second are the three methods available on the Object class (`notify, notifyAll, wait`).

To support the `synchronized` functionality, JPype defines a method called `synchronized(obj)` to be used with the Python `with` statement, where obj has to be a Java object. The return value is a monitor object that will keep the synchronization on as long as the object is kept alive. For example,

```
from jpype import synchronized

mySharedList = java.util.ArrayList()

# Give the list to another thread that will be adding items
otherThread,setList(mySharedList)

# Lock the list so that we can access it without interference
with synchronized(mySharedList):
   if not mySharedList.isEmpty():
      ... # process elements
# Resource is unlocked once we leave the block
```

The Python `with` statement is used to control the scope. Do not hold onto the monitor without a `with` statement. Monitors held outside of a `with` statement will not be released until they are broken when the monitor is garbage collected.

The other synchronization methods are available as-is on any Java object. However, as general rule one should not use synchronization methods on Java String as internal string representations may not be complete objects.

For synchronization that does not have to be shared with Java code, use Python's support directly rather than Java's synchronization to avoid unnecessary overhead.

Threading examples

Java provides a very rich set of threading tools. This can be used in Python code to extend many of the benefits of Java into Python. However, as Python has a global lock, the performance of Java threads while using Python is not as good as native Java code.

Limiting execution time

We can combine proxies and threads to produce achieve a number of interesting results. For example:

```
def limit(method, timeout):
""" Convert a Java method to asynchronous call with a specified timeout. """
    def f(*args):
         @jpype.JImplements(java.util.concurrent.Callable)
         class g:
             @jpype.JOverride
             def call(self):
                 return method(*args)
         future = java.util.concurrent.FutureTask(g())
         java.lang.Thread(future).start()
         try:
             timeunit = java.util.concurrent.TimeUnit.MILLISECONDS
             return future.get(int(timeout*1000), timeunit)
         except java.util.concurrent.TimeoutException as ex:
             future.cancel(True)
         raise RuntimeError("canceled", ex)
    return f

    print(limit(java.lang.Thread.sleep, timeout=1)(200))
    print(limit(java.lang.Thread.sleep, timeout=1)(20000))
```

Here we have limited the execution time of a Java call.

Multiprocessing

Because only one JVM can be started per process, JPype cannot be used with processes created with `fork`. Forks copy all memory including the JVM. The copied JVM usually will not function properly thus JPype cannot support multiprocessing using fork.

To use multiprocessing with JPype, processes must be created with "spawn". As the multiprocessing context is usually selected at the start and the default for Unix is fork, this requires the creating the appropriate spawn context. To launch multiprocessing properly the following recipe can be used.

```
import multiprocessing as mp

ctx = mp.get_context("spawn")
process = ctx.Process(...)
queue = ctx.Queue()
# ...
```

When using multiprocessing, Java objects cannot be sent through the default Python `Queue` methods as calls pickle without any Java support. This can be overcome by wrapping Python `Queue` to first encode to a byte stream using the JPickle package. By wrapping a `Queue` with the Java pickler any serializable Java object can be transferred between processes.

In addition, a standard Queue will not produce an error if is unable to pickle a Java object. This can cause deadlocks when using multiprocessing IPC, thus wrapping any Queue is required.

1.2.10 Miscellaneous topics

This chapter contains all the stuff that did not fit nicely into the narrative about JPype. Topics include code completion, performance, debugging Java within JPype, debugging JNI and other JPype failures, how caller sensitive methods are dealt with, and finally limitations of JPype.

Autopep8

When Autopep8 is applied a Python script, it reorganizes the imports to conform to E402. This has the unfortunate side effect of moving the Java imports above the startJVM statement. This can be avoided by either passing in `--ignore E402` or setting the ignore in `.pep8`.

Example:

```
import jpype
import jpype.imports

jpype.startJVM()

from gov.llnl.math import DoubleArray
```

Result without `--ignore E402`

```
from gov.llnl.math import DoubleArray  # Fails, no JVM running
import jpype
import jpype.imports

jpype.startJVM()
```

Performance

JPype uses JNI, which is well known in the Java world as not being the most efficient of interfaces. Further, JPype bridges two very different runtime environments, performing conversion back and forth as needed. Both of these can impose performance bottlenecks.

JNI is the standard native interface for most, if not all, JVMs, so there is no getting around it. Down the road, it is possible that interfacing with CNI (GCC's java native interface) may be used. Right now, the best way to reduce the JNI cost is to move time critical code over to Java.

Follow the regular Python philosophy : **Write it all in Python, then write only those parts that need it in C.** Except this time, it's write the parts that need it in Java.

Everytime an object is passed back and forth, it will incure a conversion cost.. In cases where a given object (be it a string, an object, an array, etc . . .) is passed often into Java, the object should be converted once and cached. For most situations, this will address speed issues.

To improve speed issues, JPype has converted all of the base classes into CPython. This is a very significant speed up over the previous versions of the module. In addition, JPype provides a number of fast buffer transfer methods. These routines are triggered automatically working with any buffer aware class such as those in NumPy.

As a final note, while a JPype program will likely be slower than its pure Java counterpart, it has a good chance of being faster than the pure Python version of it. The JVM is a memory hog, but does a good job of optimizing code execution speeds.

Code completion

Python supports a number of different code completion engines that are integrated in different Python IDEs. JPype has been tested with both the IPython greedy completion engine and Jedi. Greedy has the disadvantage that is will execute code resulting potentially resulting in an undesirable result in Java.

JPype is Jedi aware and attempts to provide whatever type information that is available to Jedi to help with completion tasks. Overloaded methods are opaque to Jedi as the return type cannot be determined externally. If all of the overloads have the same return type, the JPype will add the return type annotation permitting Jedi to autocomplete through a method return.

For example:

```
JString("hello").substring.__annotations__
# Returns {'return': <java class 'java.lang.String'>}
```

Jedi can manually be tested using the following code.

```
js = JString("hello")
src = 'js.s'
script = jedi.Interpreter(src, [locals()])
compl = [i.name for i in script.completions()]
```

This will produce a list containing all method and field that begin with the letter "s".

JPype has not been tested with other autocompletion engines such as Kite.

Garbage collection

Garbage collection (GC) is supposed to make life easier for the programmer by removing the need to manually handle memory. For the most part it is a good thing. However, just like running a kitchen with two chiefs is a bad idea, running with two garbage collections is also bad. In JPype we have to contend with the fact that both Java and Python provide garbage collection for their memory and neither provided hooks for interacting with an external garbage collector.

For example, Python is creating a bunch a handles to Java memory for a period of time but they are in a structure with a reference loop internal to Python. The structures and handles are small so Python doesn't see an issue, but each of those handles is holding 1M of memory in Java space. As the heap fills up Java begins garbage collecting, but the resources can't be freed because Python hasn't cleanup up these structures. The reverse occurs if a proxy has any large NumPy arrays. Java doesn't see a problem as it has plenty of space to work in but Python is running its GC like mad trying to free up space to work.

To deal with this issue, JPype links the two garbage collectors. Python is more aggressive in calling GC than Java and Java is much more costly than Python in terms of clean up costs. So JPype manages the balance. JPype installs a

sentinel object in Java. Whenever that sentinel is collected Java is running out of space and Python is asked to clean up its space as well. The reverse case is more complicated as Python can't just call Java's expensive routine any time it wants. Instead JPype maintains a low-water and high-water mark on Python owned memory. Each time it nears a high-water mark during a Python collection, Java GC gets called. If the water level shrinks than Java was holding up Python memory and the low-water mark is reset. Depending on the amount of memory being exchanged the Java GC may trigger as few as once every 50 Python GC cycles or as often as every other. The sizing on this is dynamic so it should scale to the memory use of a process.

Using JPype for debugging Java code

One common use of JPype is to function as a Read-Eval-Print Loop for Java. When operating Java though Python as a method of developing or debugging Java there are a few tricks that can be used to simplify the job. Beyond being able to probe and plot the Java data structures interactively, these methods include:

1) Attaching a debugger to the Java JVM being run under JPype.

2) Attaching debugging information to a Java exception.

3) Serializing the state of a Java process to be evaluated at a later point.

We will briefly discuss each of these methods.

Attaching a Debugger

Interacting with Java through a shell is great, but sometimes it is necessary to drop down to a debugger. To make this happen we need to start the JVM with options to support remote debugging.

We start the JVM with an agent which will provide a remote debugging port which can be used to attach your favorite Java debugging tool. As the agent is altering the Java code to create additional debugging hooks, this process can introduce additional errors or alter the flow of the code. Usually this is used by starting the JVM with the agent, placing a pause marker in the Python code so that developer can attach the Java debugger, executing the Python code until it hits the pause, attaching the debugger, setting break point in Java, and then asking Python to proceed.

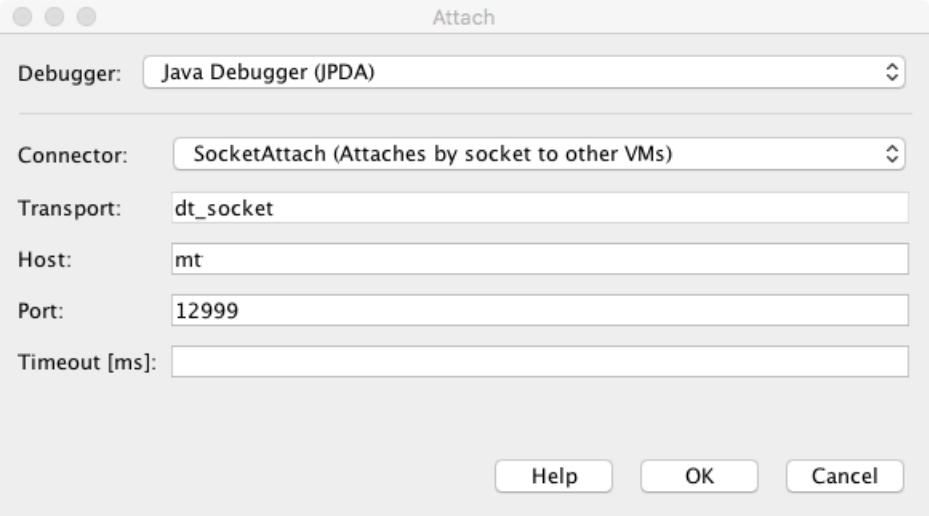
So lets flesh out the details of how to accomplish this...

```
jpype.startJVM("-Xint", "-Xdebug", "-Xnoagent",
  "-Xrunjdwp:transport=dt_socket,server=y,address=12999,suspend=n")
```

Next, add a marker in the form of a pause statement at the location where the debugger should be attached.

```
input("pause to attach debugger")
myobj.callProblematicMethod()
```

When Python reaches that point during execution, switch to a Java IDE such as NetBeans and select Debug : Attach Debugger. This brings up a window (see example below). After attaching (and setting desired break points) go back to Python and hit enter to continue. NetBeans should come to the foreground when a breakpoint is hit.



Attach data to an Exception

Sometimes getting to the level of a debugger is challenging especially if the code is large and error occurs rarely. In this case, it is often beneficial to attach data to an exception. To achieve this, we need to write a small utility class. Java exceptions are not strictly speaking expandable, but they can be chained. Thus, it we create a dummy exception holding a `java.util.Map` and attach it to as the cause of the exception, it will be passed back down the call stack until it reaches Python. We can then use `getCause()` to retrieve the map containing the relevant data.

Capturing the state

If the program is not running in an interactive shell or the program run time is long, we may not want to deal with the problem during execution. In this case, we can serialize the state of the relevant classes and variables. To use this option, we mus make sure all of the classes in Java that we are using are `Serializable`. Then add a condition that detects the faulty algorithm state. When the fault occurs, create a `java.util.HashMap` and populate it with the values to be examined from within Python. Use serialization to write the entire structure to a file. Execute the program and collect all of the state files.

Once the state files have been collected, start Python with an interactive shell and launch JPype with a classpath for the jars. Finally, deserialize the state files to access the Java structures that have been recorded.

Getting additional diagnostics

For the most part JPype does what its told, but that does not mean that there are no bugs. With some many different interactions between Python and Java there is always some untested edge-cases.

JPype has a few diagnostic tools to help deal with these sorts of problems but each of them require accessing a "private" JPype symbol which may be altered, removed, folded, spindled, or mutilated in any future release. Thus none of the following should be used in production code.

Checking the type of a cast

Sometimes it is difficult to understand why a particular method overload is being selected by the method dispatch. To check the match type for a conversion call the private method `Class._canConvertToJava`. This will produce a string naming the type of conversion that will be performed as one of `none`, `explicit`, `implicit`, or `exact`..

To test the result of the conversion process, call `Class._convertToJava`. Unlike an explicit cast, this just attempts to perform the conversion without bypassing all of the other logic involved in casting. It replicates the exact process used when a method is called or a field is set.

C++ Exceptions in JPype

Internally JPype can generate C++ exception which is converted into Python exceptions for the user. To trace an error back to its C++ source, it is necessary to obtain the original C++ exception. As all sensitive block have function names compiled in to the try catch blocks, these C++ exception stack frames can be extracted as the "cause" of a Python exception. To enable C++ stack traces use the command `_jpype.enableStacktraces(True)`. Once executed all C++ exceptions that fell through a C++ exception handling block will produce an augmented C++ stack trace. If the JPype source code is available to Python, it can even print out each line where the stack frame was caught. This is usually at the end of each function that was executed. JPype does not need to be recompiled to use this option.

Tracing

To debug a problem that resulted from a stateful interaction of elements the use of the JPype tracing mode may helpful. To enable tracing recompile JPype with the `--enable-tracing` mode set. When code is executed with tracing, every JNI call along with the object addresses and exceptions will be printed to the console. This is keyed to macros that appear at the start and end of each JPype function. These macros correspond to a try catch block.

This will often produce very large and verbose tracing logs. However, tracing is often the only way to observe a failure that originated in one JNI call but did not fail until many calls later.

Instrumentation

In order to support coverage tools, JPype can be compiled with a special instrumentation mode in which the private module command `_jpype.fault` can be used to trigger an error. The argument to the fault must be a function name as given in the `JP_TRACE_IN` macro at the start of each JPype function or a special trigger point defined in the code. When the fault point is encounter it will trigger a `SystemError`. This mode of operation can be used to replicate the path that a particular call took and verify that the error handling from that point back to Java is safe.

Because instrumentation uses the same control hooks as tracing, only one mode can be active at a time. Enabling instrumentation requires recompiling the JPype module with `--enable-coverage` option.

Using a debugger

If there is a crash in the JPype module, it may be necessary to get a backtrace using a debugger. Unfortunately Java makes this task a bit complicated. As part of its memory handling routine, Java takes over the segmentation fault handler. Whenever the fault is triggered, Java checks to see if it was the result the growth of an internal structure. If it was simply a need for additional space, Java handles the exception by allocating addition memory. On the other hand, if a fault was triggered by some external source, Java constructs a JVM fault report and then transfers control back to the usual segmentation fault handler. Java will often corrupt the stack frame. Any debugger attempting to unpack the corrupted core file will instead get random function addresses.

The alternative is for the user to start JPype with an interactive debugger and execute to the fault point. But this option also presents challenges. The first action after starting the JVM is a test to see if its segmentation fault handler was installed properly. Thus it will trigger an intentional segmentation fault. The debugger can not recognize the difference between an intentional test and an actual fault, so the test will stop the debugger. To avoid this problem debuggers such as gdb must be set to ignore the first segmentation fault. Further details on this can be found in the developer guide.

Caller sensitive methods

The Java security model tracks what caller requested the method as a means to determine the level of access to provide. Internal callers are provided privileged access to perform unsafe operations and external callers are given safer and more restricted access. To perform this task, the JVM seaches the call stack to obtain the calling methods module.

This presents a difficulty for method invoked from JNI. A method called from JNI lacks any call stack to unravel. Rather than relegating the call to a safer level of access, the security model would outright deny access to certain JPype calls. This resulted in a number of strange behaviors over the years that were forced to be worked around. This issue was finally solved with the release of Java 12 when they outright broken all calls to getMethod by throwing a NullPointer exception whenever the caller frame was not found. This inadvertent clued us into why Java would act so strangely for certain calls such as constructing a SQL database or attempting to call `Class.forName`. By creating an actual test case to work around we were able to resolve this limitation.

Once we identified the issue, the workaround is only call caller sensitive methods from within Java. But given that we call methods through JNI and the JNI interface defines no way to specify an origin for the call, the means we needed to develop an alternative calling mechanism. Instead of calling methods directly, we instead pass the method id and the list of desired arguments to the internal `org.jpype` Java package. This package unpacks the request and executes the desired method from within Java. The call stack will indicate the caller is an external jar and be given the safe and restricted level of access. The result is then passed back to through the JNI layer.

This special calling mechanism is slower and more indirect than the normal calling procedure, so its use is limited to only those methods that really require a caller sensitive procedure. The mechanism to determine which methods are caller sensitive depends on the internals of Java and have changed with Java versions. Older Java versions did not directly mark the caller sensitive methods and we must instead blanket bomb all methods belonging to `java.lang.Class`, `java.lang.ClassLoader`, and `java.sql.DriverManager`. Newer versions specifically annotate the methods requiring caller sensitive treatment, but for some reason this annotation is a package private and thus we must search through method annotations by name to find the caller sensitive annotation. Fortunately, this process is only performed once when the class is created, and very few methods have a large number of annotations so this isn't a performance hit.

JPype Known limitations

This section lists those limitations that are unlikely to change, as they come from external sources.

Restarting the JVM

JPype caches many resources to the JVM. Those resource are still allocated after the JVM is shutdown as there are still Python objects that point to those resources. If the JVM is restarted, those stale Python objects will be in a broken state and the new JVM instance will obtain the references to these resulting in a memory leak. Thus it is not possible to start the JVM after it has been shut down with the current implementation.

Running multiple JVM

JPype uses the Python global import module dictionary, a global Python to Java class map, and global JNI TypeManager map. These resources are all tied to the JVM that is started or attached. Thus operating more than one JVM does not appear to be possible under the current implementation. Further, as of Java 1.2 there is no support for creating more than one JVM in the same process.

Difficulties that would need to be overcome to remove this limitation include:

- Finding a JVM that supports multiple JVMs running in the same process. This can be achieved on some architectures by loading the same shared library multiple times with different names.

- Alternatively as all available JVM implementations support on one JVM instance per process, a communication layer would have to proxy JNI class from JPype to another process. But this has the distinct problem that remote JVMs cannot register native methods nor share memory without considerable effort.
- Which JVM would a static class method call. The class types would need to be JVM specific (ie. `JClass('org.MyObject', jvm=JVM1)`)
- How would a wrapper from two different JVM coexist in the `jpype._jclass` module with the same name if different class is required for each JVM.
- How would the user specify which JVM a class resource is created in when importing a module.
- How would objects in one JVM be passed to another.
- How can boxed and String types hold which JVM they will box to on type conversion.

Thus it appears prohibitive to support multiple JVMs in the JPype class model.

Errors reported by Python fault handler

The JVM takes over the standard fault handlers resulting in unusual behavior if Python handlers are installed. As part of normal operations the JVM will trigger a segmentation fault when starting and when interrupting threads. Pythons fault handler can intercept these operations and interpret these as real faults. The Python fault handler with then reporting extraneous fault messages or prevent normal JVM operations. When operating with JPype, Python fault handler module should be disabled.

This is particularly a problem for running under pytest as the first action it performs is to take over the error handlers. This can be disabled by adding this block as a fixture at the start of the test suite.

```
try:
    import faulthandler
    faulthandler.enable()
    faulthandler.disable()
except:
   pass
```

This code enables fault handling and then returns the default handlers which will point back to those set by Java.

Unsupported Python versions

Python 3.4 and earlier

The oldest version of Python that we currently support is Python 3.5. Before Python 3.5 there were a number of structural difficulties in the object model and the buffering API. In principle, those features could be excised from JPype to extend support to older Python 3 series version, but that is unlikely to happen without a significant effort.

Python 2

CPython 2 support was removed starting in 2020. Please do not report to us that Python 2 is not supported. Python 2 was a major drag on this project for years. Its object model is grossly outdated and thus providing for it greatly impeded progress. When the life support was finally pulled on that beast, I like many others breathed a great sigh of relief and gladly cut out the Python 2 code. Since that time JPype operating speed has improved anywhere from 300% to 10000% as we can now implement everything back in CPython rather than band-aiding it with interpreted Python code.

PyPy

The GC routine in PyPy 3 does not play well with Java. It runs when it thinks that Python is running out of resources. Thus a code that allocates a lot of Java memory and deletes the Python objects will still be holding the Java memory until Python is garbage collected. This means that out of memory failures can be issued during heavy operation. We have addressed linking the garbage collectors between CPython and Java, but PyPy would require a modified strategy.

Further, when we moved to a completely Python 3 object model we unfortunately broke some of the features that are different between CPython and PyPy. The errors make absolutely no sense to me. So unless a PyPy developer generously volunteering time for this project, this one is unlikely to happen.

Jython Python

If for some reason you wandered here to figure out how to use Java from Jython using JPype, you are clearly in the wrong place. On the other hand, if you happen to be a Jython developer who is looking for inspiration on how to support a more JPype like API that perhaps we can assist you. Jython aware Python modules often mistake JPype for Jython at least up until the point that differences in the API triggers an error.

Unsupported Java virtual machines

The open JVM implementations *Cacao* and *JamVM* are known not to work with JPype.

Unsupported Platforms

Some platforms are problematic for JPype due to interactions between the Python libraries and the JVM implementation.

Cygwin

Cygwin was usable with previous versions of JPype, but there were numerous issues for which there is was not good solution solution.

Cygwin does not appear to pass environment variables to the JVM properly resulting in unusual behavior with certain windows calls. The path separator for Cygwin does not match that of the Java DLL, thus specification of class paths must account for this. Threading between the Cygwin libraries and the JVM was often unstable.

1.3 Java QuickStart Guide

This is a quick start guide to using JPype with Java. This guide will show a series of snippets with the corresponding commands in both Java and Python for using JPype. The *JPype User Guide* and *API Reference* have additional details on the use of the JPype module.

JPype uses two factory classes (`JArray` and `JClass`) to produce class wrappers which can be used to create all Java objects. These serve as both the base class for the corresponding hierarchy and as the factory to produce new wrappers. Casting operators are used to construct specify types of Java types (`JObject`, `JString`, `JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`). Two special classes serve as the base classes for exceptions (`JException`) and interfaces (`JInterface`). There are a small number of support methods to help in controlling the JVM. Lastly, there are a few annotations used to create customized wrappers.

For the purpose of this guide, we will assume that the following classes were defined in Java. We will also assume the reader knows enough Java and Python to be dangerous.

```
package org.pkg;

publc class BassClass
{
   public callMember(int i)
   {}
}

public class MyClass extends BaseClass
{
   final public static int CONST_FIELD = 1;
   public static int staticField = 1;
   public int memberField = 2;
   int internalField =3;

   public MyClass() {}
   public MyClass(int i) {}

   public static void callStatic(int i) {}
   public void callMember(int i) {}

   // Python name conflict
   public void pass() {}

   public void throwsException throws java.lang.Exception {}

   // Overloaded methods
   public call(int i) {}
   public call(double d) {}
}
```

1.3.1 Starting JPype

The hardest thing about using JPype is getting the jars loaded into the JVM. Java is curiously unfriendly about reporting problems when it is unable to find a jar. Instead, it will be reported as an `ImportError` in Python. These patterns will help debug problems with jar loading.

Once the JVM is started Java packages that are within a top level domain (TLD) are exposed as Python modules allowing Java to be treated as part of Python.

Description	Java	Python

Start Java Virtual Machine (JVM)		`# Import module`
		`import jpype`
		
		`# Enable Java imports`
		`import jpype.imports`
		
		`# Pull in types`
		`from jpype.types import *`
		
		`# Launch the JVM`
		`jpype.startJVM()`

Start Java Virtual Machine (JVM) with a classpath		`# Launch the JVM` `jpype.startJVM(classpath ↪= ['jars/*'])`
Import default Java namespace[1]		`import java.lang`
Add a set of jars from a directory[2]		`jpype.addClassPath("/my/ ↪path/*")`
Add a specific jar to the classpath[2]		`jpype.addClassPath('/my/ ↪path/myJar.jar')`
Print JVM CLASSPATH[3]		`from java.lang import ↪System` `print(System.getProperty( ↪"java.class.path"))`

1.3.2 Classes/Objects

Java classes are presented wherever possible similar to Python classes. The only major difference is that Java classes and objects are closed and cannot be modified. As Java is strongly typed, casting operators are used to select specific overloads when calling methods. Classes are either imported using a module, loaded using `JPackage` or loaded with the `JClass` factory.

[1] All `java.lang.*` classes are available.
[2] Must happen prior to starting the JVM
[3] After JVM is started

Description	Java	Python
Import a class[4]	`import org.pkg.MyClass`	`from org.pkg import MyClass`
Import a class and rename[4]		`from org.pkg import MyClass as OurClass`
Import multiple classes from a package[5]		`from org.pkg import MyClass, AnotherClass`
Import a java package for long name access[6]		`import org.pkg`
Import a class static[7]	`import org.pkg.MyClass.CONST_FIELD`	`from org.pkg.MyClass import CONST_FIELD`
Import a class without tld[8]	`import zippy.NonStandard`	`NonStandard = JClass('zippy.NonStandard')`
Construct an object	`MyClass myObject = new MyClass(1);`	`myObject = MyClass(1)`
Constructing a cless with full class name		`import org.pkg` `myObject = org.pkg.MyClass(args)`
Get a static field	`int var = MyClass.staticField;`	`var = MyClass.staticField`
Get a member field	`int var = myObject.memberField;`	`var = myObject.memberField`
Set a static field[9]	`MyClass.staticField = 2;`	`MyClass.staticField = 2`
Set a member field[9]	`myObject.memberField = 2;`	`myObject.memberField = 2`
Call a static method	`MyClass.callStatic(1);`	`MyClass.callStatic(1)`
Call a member method	`myObject.callMember(1);`	`myObject.callMember(1)`
Access member with Python naming conflict[10]	`myObject.pass()`	`myObject.pass_()`
Checking inheritance		
	`if (obj instanceof MyClass) {...}`	`if isinstance(obj, MyClass): ...`
Checking if Java class wrapper		

1.3.3 Exceptions

Java exceptions extend from Python exceptions and can be dealt with in the same way as Python native exceptions. JException serves as the base class for all Java exceptions.

Description	Java	Python
Catch an exception	`try { myObject.throwsException(); } catch (java.lang.Exception ex) { ... }`	`try: myObject.throwsException() except java.lang.Exception as ex: ...`
Throw an exception to Java	`throw new java.lang.Exception( "Problem");`	`raise java.lang.Exception( "Problem")`
Checking if Java exception wrapper		`if (isinstance(obj, JException)): ...`
Closeable items	`try (InputStream is = Files.newInputStream(file)) { ... }`	`with Files.newInputStream(file) as is: ...`

1.3.4 Primitives

Most Python primitives directly map into Java primitives. However, Python does not have the same primitive types, and it is necessary to cast to a specific Java primitive type whenever there are Java overloads that would otherwise be in conflict. Each of the Java types are exposed in JPype (`JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`).

[4] This will report an error if the class is not found.
[5] This will report an error if the classes are not found.
[6] Does not report errors if the package is invalid.
[7] Constants, static fields, and static methods can be imported.
[8] `JClass` loads any class by name including inner classes.
[9] This produces an error for final fields.
[10] Underscore is added during wrapping.

Description	Java	Python
Casting to hit an overload[11]	`myObject.call((int)v);`	`myObject.call(JInt(v))`
Create a primitive array	`int[] array = new int[5]`	`array = JArray(JInt)(5)`
Create an initialized primitive array[12]	`int[] array = new int[]{1, ↪2,3}`	`array = JArray(JInt)([1,2, ↪3])`
Put a specific primitive type on a list	`List<Integer> myList = new ArrayList<>(); myList.add(1);`	`from java.util import␣ ↪ArrayList myList = ArrayList() myList.add(JInt(1))`
Boxing a primitive[13]	`Integer boxed = 1;`	`boxed = JObject(JInt(1))`

1.3.5 Strings

Java strings are similar to Python strings. They are both immutable and produce a new string when altered. Most operations can use Java strings in place of Python strings, with minor exceptions as Python strings are not completely duck typed. When comparing or using as dictionary keys, all JString objects should be converted to Python.

[11] `JInt` acts as a casting operator
[12] list, sequences, or np.array can be used to initialize.
[13] `JInt` specifies the prmitive type. `JObject` boxes the primitive.

Description	Java	Python
Create a Java string[14]	`String javaStr = new String("foo");`	`myStr = JString("foo")`
Create a Java string from bytes[15]	`byte[] b;` `String javaStr = new String(b, "UTF-8");`	`b= b'foo'` `myStr = JString(b, "UTF-8")`
Converting Java string		`str(javaStr)`
Comparing Python and Java strings[16]		`str(javaStr) == pyString`
Comparing Java strings	`javaStr.equals("foo")`	`javaStr == "foo"`
Checking if java string		`if (isinstance(obj, JString): ...`

1.3.6 Arrays

Arrays are create using the JArray class factory. They operate like Python lists, but they are fixed in size.

[14] `JString` constructs a `java.lang.String`
[15] All `java.lang.String` constuctors work.
[16] `str()` converts the object for comparison

Description	Java	Python
Create a single dimension array	`MyClass[] array = new MyClass[5];`	`array = JArray(MyClass)(5)`
Create a multi dimension array	`MyClass[][] array2 = new MyClass[5][];`	`array2 = JArray(MyClass, 2)(5)`
Access an element	`array[0] = new MyClass()`	`array[0] = MyClass()`
Size of an array	`array.length`	`len(array)`
Convert to Python list		`pylist = list(array)`
Iterate elements	`for (MyClass element: array) {...}`	`for element in array: ...`
Checking if java array wrapper		`if (isinstance(obj, JArray): ...`

1.3.7 Collections

Java standard containers are available and are overloaded with Python syntax where possible to operate in a similar fashion to Python objects.

Description	Java	Python
Import list type	`import java.util.ArrowList;`	`from java.util import ArrowList`
Construct a list	`List<Integer> myList=new ArrowArrayList<>();`	`myList=ArrayList()`
Get length of list	`int sz = myList.size();`	`sz = len(myList)`
Get list item	`Integer i = myList.get(0)`	`i = myList[0]`
Set list item[17]	`myList.set(0, 1)`	`myList[0]=Jint(1)`
Iterate list elements	`for (Integer element: ArrowmyList) {...}`	`for element in myList: ...`
Import map type	`import java.util.HashMap;`	`from java.util import ArrowHashMap`
Construct a map	`Map<String,Integer> ArrowmyMap=new HashMap<>();`	`myMap=HashMap()`
Get length of map	`int sz = myMap.size();`	`sz = len(myMap)`
Get map item	`Integer i = myMap.get("foo ArrowArrow")`	`i = myMap["foo"]`
Set map item[17]	`myMap.set("foo", 1)`	`myMap["foo"]=Jint(1)`
Iterate map entries	`for (Map.Entry<String, ArrowInteger> e : myMap.entrySet()) {...}`	`for e in myMap.entrySet(): ...`

[17] Casting is required to box primitives to the correct type.

1.3.8 Reflection

Java reflection can be used to access operations that are outside the scope of the JPype syntax. This includes calling a specific overload or even accessing private methods and fields.

Description	Java	Python
Access Java reflection class	`MyClass.class`	`MyClass.class_`

Access a private field by name[18]		`cls = myObject.class_`
		`field = cls.`
		`↪getDeclaredField(`
		`  "internalField")`
		`field.setAccessible(True)`
		`field.get()`

Accessing a specific overload[19]		`cls = MyClass.class_`
		`cls.getDeclaredMethod(`
		`↪"call", JInt)`
		`cls.invoke(myObject,␣`
		`↪JInt(1))`

Convert a `java.lang.Class` into Python wrapper[20]		`# Something returned a␣`
		`↪java.lang.Class`
		`MyClassJava =␣`
		`↪getClassMethod()`
		
		`# Convert to it to Python`
		`MyClass =␣`
		`↪JClass(myClassJava)`

Load a class with a external class loader	`ClassLoader cl`
	`  = new␣`
	`↪ExternalClassLoader();`
	`Class cls`
	`  = Class.forName(`
	`↪"External",`
	`  True,␣`
	`↪cl)`
		`cl = ExternalClassLoader()`
		`cls = JClass("External",␣`
		`↪loader=cl)`

Accessing base method implementation		`from org.pkg import \`
		`  BaseClass, MyClass`
		`myObject = MyClass(1)`
		`BaseClass.`
		`↪callMember(myObject, 2)`

[18] This is prohibited after Java 8
[19] types must be exactly specified.
[20] Rarely required unless the class was supplied external such as generics.

1.3.9 Implements and Extension

JPype can implement a Java interface by annotating a Python class. Each method that is required must be implemented.

JPype does not support extending a class directly in Python. Where it is necessary to exend a Java class, it is required to create a Java extension with an interface for each methods that are to be accessed from Python.

Description	Java	Python
Implement an interface	`public class PyImpl implements MyInterface { public void call() {...} }`	`@JImplements(MyInterface) class PyImpl(object): @JOverride def call(self): pass`
Extending classes[21]		None
Lambdas[21]		None

Don't like the formatting? Feel the guide is missing something? Submit a pull request at the project page.

1.4 API Reference

1.4.1 JVM Functions

These functions control and start the JVM.

jpype.**startJVM**(**args*, ***kwargs*)

Starts a Java Virtual Machine. Without options it will start the JVM with the default classpath and jvmpath.

The default classpath is determined by `jpype.getClassPath()`. The default jvmpath is determined by `jpype.getDefaultJVMPath()`.

Parameters ***args** (*Optional, str[]*) – Arguments to give to the JVM. The first argument may be the path the JVM.

Keyword Arguments

- **jvmpath** (*str*) – Path to the jvm library file, Typically one of (`libjvm.so`, `jvm.dll`, ...) Using None will apply the default jvmpath.
- **classpath** (*str, [str]*) – Set the classpath for the jvm. This will override any classpath supplied in the arguments list. A value of None will give no classpath to JVM.
- **ignoreUnrecognized** (*bool*) – Option to JVM to ignore invalid JVM arguments. Default is False.
- **convertStrings** (*bool*) – Option to JPype to force Java strings to cast to Python strings. This option is to support legacy code for which conversion of Python strings was the default. This will globally change the behavior of all calls using strings, and a value of True is NOT recommended for newly developed code.

  The default value for this option during 0.7 series is True. The option will be False starting in 0.8. A warning will be issued if this option is not specified during the transition period.

[21] Support for use of Python function as Java 8 lambda is WIP.

Raises

- `OSError` – if the JVM cannot be started or is already running.
- `TypeError` – if an invalid keyword argument is supplied or a keyword argument conflicts with the arguments.

`jpype.`**`shutdownJVM`**`()`
Shuts down the JVM.

This method shuts down the JVM and thus disables access to existing Java objects. Due to limitations in the JPype, it is not possible to restart the JVM after being terminated.

`jpype.`**`getDefaultJVMPath`**`()`
Retrieves the path to the default or first found JVM library

Returns The path to the JVM shared library file

Raises

- `JVMNotFoundException` – If there was no JVM found in the search path.
- `JVMNotSupportedException` – If the JVM was found was not compatible with Python due to cpu architecture.

`jpype.`**`getClassPath`**`(`*`env=True`*`)`
Get the full java class path.

Includes user added paths and the environment CLASSPATH.

Parameters **`env`** (*`Optional, bool`*) – If true then environment is included. (default True)

1.4.2 Class importing

JPype supports several styles of importing. The newer integrated style is provided by the *imports* module. The older `JPackage` method is available for accessing package trees with less error checking. Direct loading of Java classes can be made with *JClass*.

For convenience, the JPype module predefines the following `JPackage` instances for `java` and `javax`.

`class` `jpype.`**`JPackage`**`(`*`name`*, *`strict=False`*, *`pattern=None`*`)`
Gateway for automatic importation of Java classes.

This class allows structured access to Java packages and classes. This functionality has been replaced by `jpype.imports`, but is still useful in some cases.

Only the root of the package tree need be declared with the `JPackage` constructor. Sub-packages will be created on demand.

For example, to import the w3c DOM package:

```
Document = JPackage('org').w3c.dom.Document
```

Under some situations such as a missing jar the resulting object will be a JPackage object rather than the expected java class. This results in rather challanging debugging messages. Thus the `jpype.imports` module is preferred. To prevent these types of errors a package can be declares as `strict` which prevents expanding package names that do not comply with Java package name conventions.

Parameters

- **`path`** (*`str`*) – Path into the Java class tree.

- **strict** (*bool, optional*) – Requires Java paths to conform to the Java package naming convention. If a path does not conform and a class with the required name is not found, the AttributeError is raise to indicate that the class was not found.

Example

```
# Alias into a library
google = JPackage("com.google")

# Access members in the library
result = google.common.IntMath.pow(x,m)
```

1.4.3 Class Factories

class jpype.**JClass**

Meta class for all java class instances.

JClass when called as an object will contruct a new java Class wrapper.

All python wrappers for java classes derived from this type. To test if a python class is a java wrapper use `isinstance(obj, jpype.JClass)`.

Parameters **className** (*str*) – name of a java type.

Keyword Arguments

- **loader** (*java.lang.ClassLoader*) – specifies a class loader to use when creating a class.
- **initialize** (*bool*) – Passed to class loader when loading a class using the class loader.

Returns a new wrapper for a Java class

Return type JavaClass

Raises `TypeError` – if the component class is invalid or could not be found.

class jpype.**JArray**(**args*, ***kwargs*)

Create a java array class for a Java type of a given dimension.

This serves as a base type and factory for all Java array classes. The resulting Java array class can be used to construct a new array with a given size or members.

JPype arrays support Python operators for iterating, length, equals, not equals, subscripting, and limited slicing. They also support Java object methods, clone, and length property. Java arrays may not be resized, thus elements cannot be added nor deleted. Currently, applying the slice operator produces a new Python sequence.

Example

```
# Define a new array class for ``int[]``
IntArrayCls = JArray(JInt)

# Create an array holding 10 elements
#   equivalent to Java ``int[] x=new int[10]``
x = IntArrayCls(10)

# Create a length 3 array initialized with [1,2,3]
```

(continues on next page)

(continued from previous page)

```
#   equivalent to Java ``int[] x = new int[]{1,2,3};``
x = IntArrayCls([1,2,3])

# Operate on an array
print(len(x))
print(x[0])
print(x[:-2])
x[1:]=(5,6)

if isinstance(x, JArray):
     print("object is a java array")

if issubclass(IntArrayCls, JArray):
     print("class is a java array type.")
```

Parameters

- **javaClass** (*str, type*) – Is the type of element to hold in the array.
- **ndims** (*Optional, int*) – the number of dimensions of the array (default=1)

Returns A new Python class that representing a Java array class.

Raises `TypeError` – if the component class is invalid or could not be found.

Note: javaClass can be specified in three ways:

- as a string with the name of a java class.
- as a Java primitive type such as `jpype.JInt`.
- as a Java class type such as `java.lang.String`.

`jpype.`**JException**

1.4.4 Java Types

JPype has types for each of the Java primitives: `JBoolean`, `JByte`, `JShort`, `JInt`, `JLong`, `JFloat` and `JDouble`. There is one class for working with Java objects, `JObject`. This serves to cast to a specific object type. There is a `JString` type provided for convenience when creating or casting to strings.

class `jpype.`**JObject**(**args*, ***kwargs*)

Base class for all object instances.

It can be used to test if an object is a java object instance with `isinstance(obj, JObject)`.

Calling `JObject` as a function can be used to covert or cast to specific Java type. It will box primitive types and supports an option type to box to.

This wrapper functions four ways.

- If the no type is given the object is automatically cast to type best matched given the value. This can be used to create a boxed primitive. `JObject(JInt(i))`
- If the type is a primitve, the object will be the boxed type of that primitive. `JObject(1, JInt)`

- If the type is a Java class and the value is a Java object, the object will be cast to the Java class and will be an exact match to the class for the purposes of matching arguments. If the object is not compatible, an exception will be raised.
- If the value is a python wrapper for class it will create a class instance. This is aliased to be much more obvious as the `class_` member of each Java class.

Parameters

- **value** – The value to be cast into an Java object.
- **type** (*Optional, type*) – The type to cast into.

Raises `TypeError` – If the object cannot be cast to the specified type, or the requested type is not a Java class or primitive.

class jpype.**JString**(**args*, ***kwargs*)

Base class for `java.lang.String` objects

When called as a function, this class will produce a `java.lang.String` object. It can be used to test if an object is a Java string using `isinstance(obj, JString)`.

1.4.5 Threading

jpype.**synchronized**(*obj*)

Creates a resource lock for a Java object.

Produces a monitor object. During the lifespan of the monitor the Java will not be able to acquire a thread lock on the object. This will prevent multiple threads from modifying a shared resource.

This should always be used as part of a Python `with` startment.

Parameters **obj** – A valid Java object shared by multiple threads.

Example:

```
with synchronized(obj):
   # modify obj values

# lock is freed when with block ends
```

jpype.**isThreadAttachedToJVM**()

Checks if a thread is attached to the JVM.

Python automatically attaches threads when a Java method is called. This creates a resource in Java for the Python thread. This method can be used to check if a Python thread is currently attached so that it can be disconnected prior to thread termination to prevent leaks.

Returns True if the thread is attached to the JVM, False if the thread is not attached or the JVM is not running.

jpype.**attachThreadToJVM**()

Attaches a thread to the JVM.

The function manually connects a thread to the JVM to allow access to Java objects and methods. JPype automaticatlly attaches when a Java resource is used, so a call to this is usually not needed.

Raises `RuntimeError` – If the JVM is not running.

jpype.**detachThreadFromJVM**()
Detaches a thread from the JVM.

This function detaches the thread and frees the associated resource in the JVM. For codes making heavy use of threading this should be used to prevent resource leaks. The thread can be reattached, so there is no harm in detaching early or more than once. This method cannot fail and there is no harm in calling it when the JVM is not running.

1.4.6 Decorators

JPype uses ordinary Python classes to implement functionality in Java. Adding these decorators to a Python class will mark them for use by JPype to interact with Java classes.

1.4.7 Proxies

JPype can implement Java interfaces either by using decorators or by manually creating a JProxy. Java only support proxying interfaces, thus we cannot extend an existing Java class.

jpype.**JProxy**

1.4.8 Customized Classes

JPype provides standard customizers for Java interfaces so that Java objects have syntax matching the corresponding Python objects. The customizers are automatically bound to the class on creation without user intervention. We are documentating the functions that each customizer adds here.

These internal classes can be used as example of how to implement your own customizers for Java classes.

class jpype._jcollection.**_JIterable**
Customizer for java.util.Iterable

This customizer adds the Python iterator syntax to classes that implement Java Iterable.

class jpype._jcollection.**_JCollection**
Customizer for java.util.Collection

This customizer adds the Python functions len() and del to Java Collions to allow for Python syntax.

class jpype._jcollection.**_JList**
Customizer for java.util.List

This customizer adds the Python list operator to function on classes that implement the Java List interface.

class jpype._jcollection.**_JMap**
Customizer for java.util.Map

This customizer adds the Python list and len operators to classes that implement the Java Map interface.

class jpype._jcollection.**_JIterator**
Customizer for java.util.Iterator

This customizer adds the Python iterator concept to classes that implement the Java Iterator interface.

class jpype._jcollection.**_JEnumeration**
Customizer for java.util.Enumerator

This customizer adds the Python iterator concept to classes that implement the Java Enumerator interface.

class jpype._jio.**_JCloseable**

Customizer for `java.lang.AutoCloseable` and `java.io.Closeable`

This customizer adds support of the *with* operator to all Java classes that implement Java AutoCloseable interface.

Example:

```
from java.nio.files import Files, Paths
with Files.newInputStream(Paths.get("foo")) as fd:
  # operate on the input stream

# Input stream closes at the end of the block.
```

1.4.9 Modules

Optional JPype behavior is stored in modules. These optional modules can be imported to add additional functionality.

JPype Imports Module

Once imported this module will place the standard TLDs into the python scope. These tlds are `java`, `com`, `org`, `gov`, `mil`, `net` and `edu`. Java symbols from these domains can be imported using the standard Python syntax.

Import customizers are supported in Python 3.6 or greater.

Forms supported:

- **import <java_pkg> [as <name>]**
- **import <java_pkg>.<java_class> [as <name>]**
- **from <java_pkg> import <java_class>[,<java_class>*]**
- **from <java_pkg> import <java_class> [as <name>]**
- **from <java_pkg>.<java_class> import <java_static> [as <name>]**
- **from <java_pkg>.<java_class> import <java_inner> [as <name>]**

For further information please read the *JImport* guide.

Requires: Python 3.6 or later

Example:

```
import jpype
import jpype.imports
jpype.startJVM()

# Import java packages as modules
from java.lang import String
```

jpype.imports.**registerDomain**(*mod*, *alias=None*)

Add a java domain to python as a dynamic module.

This can be used to bind a Java path to a Python path.

Parameters

- **mod** (*str*) – Is the Python module to bind to Java.

- **alias** (*str, optional*) – Is the name of the Java path if different than the Python name.

jpype.imports.**registerImportCustomizer**(*customizer*)
Import customizers can be used to import python packages into java modules automatically.

class jpype.imports.**JImportCustomizer**
Base class for Import customizer.

Import customizers should implement canCustomize and getSpec.

Example:

```
# Site packages for each java package are stored under $DEVEL/<java_pkg>/py
class SiteCustomizer(jpype.imports.JImportCustomizer):
    def canCustomize(self, name):
        if name.startswith('org.mysite') and name.endswith('.py'):
            return True
        return False
    def getSpec(self, name):
        pname = name[:-3]
        devel = os.environ.get('DEVEL')
        path = os.path.join(devel, pname,'py','__init__.py')
        return importlib.util.spec_from_file_location(name, path)
```

JPype Pickle Module

This module contains overloaded Pickler and Unpickler classes that operate on Java classes. Pickling of Java objects is restricted to classes that implement Serializable. Mixed pickles files containing both Java and Python objects are allowed. Only one copy of each Java object will appear in the pickle file even it is appears multiple times in the data structure.

JPicklers and JUnpickler use Java ObjectOutputStream and ObjectInputStream to serial objects. All of the usual java serialization errors may be thrown.

For Python 3 series, this is backed by the native cPickler implementation.

Example:

```
myobj = jpype.JClass('java.util.ArrayList')
myobj.add("test")

from jpype.pickle import JPickler, JUnpickler
with open("test.pic", "wb") as fd:
  JPickler(fd).dump(myobj)

with open("test.pic", "rb") as fd:
  newobj = JUnpickler.load(fd)
```

Proxies and other JPype specific module resources cannot be pickled currently.

Requires: Python 3.6 or later

class jpype.pickle.**JPickler**(*file*, **args*, ***kwargs*)
Pickler overloaded to support Java objects

Parameters

- **file** – a file or other writeable object.
- ***args** – any arguments support by the native pickler.

Raises

- `java.io.NotSerializableException` – if a class is not serializable or one of its members
- `java.io.InvalidClassException` – an error occures in constructing a serialization.

class jpype.pickle.**JUnpickler**(*file*, **args*, ***kwargs*)

Unpickler overloaded to support Java objects

Parameters

- **file** – a file or other readable object.
- ***args** – any arguments support by the native unpickler.

Raises

- `java.lang.ClassNotFoundException` – if a serialized class is not found by the current classloader.
- `java.io.InvalidClassException` – if the serialVersionUID for the class does not match, usually as a result of a new jar version.
- `java.io.StreamCorruptedException` – if the pickle file has been altered or corrupted.

JPype Beans Module

This customizer finds all occurances of methods with get or set and converts them into Python properties. This behavior is sometimes useful in programming with JPype with interactive shells, but also leads to a lot of confusion. Is this class exposing a variable or is this a property added JPype. It was the default behavior until 0.7.

As an unnecessary behavior that violates both the Python principle *"There should be one– and preferably only one –obvious way to do it."* and the C++ principle *"You only pay for what you use"*. Thus this misfeature was removed from the distribution as a default. However, given that it is at times useful to have methods appear as properties, it was moved to a an optional module.

To use beans as properties:

```
import jpype.beans
```

The beans property modification is a global behavior and applies retroactively to all classes currently loaded. Once started it can never be undone.

JPype Types Module

Optional module containing only the Java types and factories used by JPype. Classes in this module include `JArray`, `JClass`, `JBoolean`, `JByte`, `JChar`, `JShort`, `JInt`, `JLong`, `JFloat`, `JDouble`, `JString`, `JObject`, and `JException`.

Example

```
from jpype.types import *
```

1.5 JImport

Module for dynamically loading Java Classes using the import system.

This is a replacement for the jpype.JPackage("com").fuzzy.Main type syntax. It features better safety as the objects produced are checked for class existence. To use java imports, import the domains package prior to importing a java class.

This module supports three different styles of importing java classes.

1.5.1 1) Import of the package path

import <java_package_path>

Importing a series of package creates a path to all classes contained in that package. It does not provide access the the contained packages. The root package is added to the global scope. Imported packages are added to the directory of the base module.

```
import java.lang      # Adds java as a module
import java.util

mystr = java.lang.String('hello')
mylist = java.util.LinkedList()
path = java.nio.files.Paths.get() # ERROR java.nio.files not imported
```

1.5.2 2) Import of the package path as a module

import <java_package> as <var>

A package can be imported as a local variable. This provides access to all java classes in that package. Contained packages are not available.

Example:

```
import java.nio as nio
bb = nio.ByteBuffer()
path = nio.file.Path()    # ERROR subpackages file must be imported
```

1.5.3 3) Import a class from an object

from <java_package> import <class>[,<class>*] [as <var>]

An individual class can be imported from a java package. This supports inner classes as well.

Example:

```
# Import one class
from java.lang import String
mystr = String('hello')

# Import multiple classes
from java.lang import Number,Integer,Double
# Import java inner class java.lang.ProcessBuilder.Redirect
from java.lang.ProcessBuilder import Redirect
```

This method can also be used to import a static variable or method from a class.

1.5.4 Import caveats

Wild card Imports

Wild card imports for classes will import all static method and fields into the global namespace. They will also import any inner classes that have been previously be accessed.

Wild card importation of package symbols are not currently supported and have unpredictable effects. Because of the nature of class loaders it is not possible to determine what classes are currently loaded. Some classes are loaded by the boot strap loader and thus are not available for discovery.

As currently implemented [from <java_package> import *] will import all classes and static variables which have already been imported by another import call. As a result which classes will be imported is based on the code pat and thus very unreliable.

It is possible to determine the classes available using Guava for java extension jars or for jars specifically loaded in the class path. But this is sufficiently unreliable that we recommend not using wildcards for any purpose.

Keyword naming

Occasionally a java class may contain a python keyword. Python keywords as automatically remapped using trailing underscore.

Example:

```
from org.raise_ import Object  => imports "org.raise.Object"
```

Controlling Java package imports

By default domains imports four top level domains (TLD) into the python import system (com, gov, java, org). Additional domains can be added by calling registerDomain. Domains can be an alias for a java package path.

Example:

```
domains.registerDomain('jname')
from jname.framework import FrameObject
domains.registerDomain('jlang', alias='java.lang')
from jlang import String
```

Limitations

- Wildcard imports are unreliable and should be avoided. Limitations in the Java specification are such that there is no way to get class information at runtime. Python does not have a good hook to prevent the use of wildcard loading.
- Non-static members can be imported but can not be called without an instance. Jpype does not provide an easy way to determine which functions objects can be called without an object.

1.6 Changelog

This changelog *only* contains changes from the *first* pypi release (0.5.4.3) onwards.

- **Next version - unreleased**

- **0.7.5 - 2020-05-10**
  - Updated docs.
  - Fix corrupt conda release.
- **0.7.4 - 4-28-2020**
  - Corrected a resource leak in arrays that affects array initialization, and variable argument methods.
  - Upgraded diagnostic tracing and JNI checks to prevent future resource leaks.
- **0.7.3 - 4-17-2020**
  - **Replaced type management system**, memory management for internal classes is now completely in Java to allow enhancements for buffer support and revised type conversion system.
  - Python module `jpype.reflect` will be removed in the next release.
  - `jpype.startJVM` option `convertStrings` default will become False in the next release.
  - Undocumented feature of using a Python type in `JObject(obj, type=tp)` is deprecated to support casting to Python wrapper types in Java in a future release.
  - Dropped support for Cygwin platform.
  - `JFloat` properly follows Java rules for conversion from `JDouble`. Floats outside of range map to inf and -inf.
  - `java.lang.Number` converts automatically from Python and Java numbers. Java primitive types will cast to their proper box type when passed to methods and fields taking Number.
  - `java.lang.Object` and `java.lang.Number` box signed, sized numpy types (int8, int16, int32, int64, float32, float64) to the Java boxed type with the same size automatically. Architecture dependent numpy types map to Long or Double like other Python types.
  - Explicit casting using primitives such as JInt will not produce an `OverflowError`. Implicit casting from Python types such as int or float will.
  - Returns for number type primitives will retain their return type information. These are derived from Python `int` and `float` types thus no change in behavior unless chaining from a Java methods which is not allowed in Java without a cast. `JBoolean` and `JChar` still produce Python types only.
  - Add support for direct conversion of multi-dimensional primitive arrays with `JArray.of(array, [dtype=type])`
  - `java.nio.Buffer` derived objects can convert to memoryview if they are direct. They can be converted to NumPy arrays with `numpy.asarray(memoryview(obj))`.
  - Proxies created with `@JImplements` properly implement `toString`, `hashCode`, and `equals`.
  - Proxies pass Python exceptions properly rather converting to `java.lang.RuntimeException`
  - `JProxy.unwrap()` will return the original instance object for proxies created with JProxy. Otherwise will return the proxy.
  - JProxy instances created with the `convert=True` argument will automatic unwrap when passed from Java to Python.
  - JProxy only creates one copy of the invocation handler per garbage collection rather than once per use. Thus proxy objects placed in memory containers will have the same object id so long as Java holds on to it.
  - `@JImplements` with keyword argument `deferred` can be started prior to starting the JVM. Methods are checked at first object creation.

- Fix bug that was causing `java.lang.Comparable`, `byte[]`, and `char[]` to be unhashable.
- Fix bug causing segfault when throwing Exceptions which lack a default constructor.
- Fixed segfault when methods called by proxy have incorrect number of arguments.
- Fixed stack overflow crash on iterating ImmutableList
- `java.util.Map` conforms to Python `collections.abc.Mapping` API.
- `java.lang.ArrayIndexOutOfBoundsException` can be caught with `IndexError` for consistency with Python exception usage.
- `java.lang.NullPointerException` can be caught with `ValueError` for consistency with Python exception usage.
- **Replaced type conversion system**, type conversions test conversion once per type improving speed and increasing flexiblity.
- User defined implicit conversions can be created with `@JConversion` decorator on Python function taking Java class and Python object. Converter function must produce a Java class instance.
- `pathlib.Path` can be implicitly converted into `java.lang.File` and `java.lang.Path`.
- `datetime.datatime` can implicitly convert to `java.time.Instant`.
- `dict` and `collections.abc.Mapping` can convert to `java.util.Map` if all element are convertable to Java. Otherwise, `TypeError` is raised.
- `list` and `collections.abc.Sequence` can convert to `java.util.Collection` if all elements are convertable to Java. Otherwise, `TypeError` is raised.

- **0.7.2 - 2-28-2020**
  - C++ and Java exceptions hold the traceback as a Python exception cause. It is no longer necessary to call stacktrace() to retrieve the traceback information.
  - Speed for call return path has been improved by a factor of 3.
  - Multidimensional array buffer transfers increase speed transfers to numpy substantially (orders of magnitude). Multidimension primitive transfers are read-only copies produced inside the JVM with C contiguous layout.
  - All exposed internals have been replaced with CPython implementations thus symbols *__javaclass__*, *__javavalue__*, and *__javaproxy__* have been removed. A dedicated Java slot has been added to all CPython types derived from *_jpype* class types. All private tables have been moved to CPython. Java types must derive from the metaclass *JClass* which enforces type slots. Mixins of Python base classes is not permitted. Objects, Proxies, Exceptions, Numbers, and Arrays derive directly from internal CPython implementations. See the ChangeLog-0.7.2 for details of all changes.
  - Internal improvements to tracing and exception handling.
  - Memory leak in convertToDirectBuffer has been corrected.

  = **Arrays slices are now a view which support writeback to the original** like numpy array. Array slices are no longer covariant returns of list or numpy.array depending on the build procedure.
  - Array slices support steps for both set and get.
  - Arrays now implement *__reversed__*
  - Incorrect mapping of floats between 0 and 1 to False in setting Java boolean array members is corrected.
  - Java arrays now properly assert range checks when setting elements from sequences.
  - Java arrays support memoryview API and no longer required NumPy to transfer buffer contents.

  - Numpy is no longer an optional extra. Memory transfer to NumPy is available without compiling for numpy support.
  - JInterface is now a meta class. Use `isinstance(cls, JInterface)` to test for interfaces.
  - Fixed memory leak in Proxy invocation
  - Fixed bug with Proxy not converting when passed as an argument to Python functions during execution of proxies
  - Missing tlds "mil", "net", and "edu" added to default imports.
  - Enhanced error reporting for UnsupportedClassVersion during startup.
  - Corrections for collection methods to improve complience with Python containers.
    * java.util.Map gives KeyError if the item is not found. Values that are `null` still return `None` as expected. Use `get()` if empty keys are to be treated as `None`.
    * java.util.Collection `__delitem__` was removed as it overloads oddly between `remove(Object)` and `remove(int)` on Lists. Use Java `remove()` method to access the original Java behavior, but a cast is strongly recommended to to handle the overload.
  - java.lang.IndexOutOfBoundsException can be caught with IndexError for complience when accessing `java.util.List` elements.

- **0.7.1 - 12-16-2019**
  - Updated the keyword safe list for Python 3.
  - Automatic conversion of CharSequence from Python strings.
  - java.lang.AutoCloseable supports Python "with" statement.
  - Hash codes for boxed types work properly in Python 3 and can be used as dictionary keys again (same as JPype 0.6). Java arrays have working hash codes, but as they are mutable should not be used as dictionary keys. java.lang.Character, java.lang.Float, and java.lang.Double all work as dictionary keys, but due to differences in the hashing algorithm do not index to the same location as Python native types and thus may cause issues when used as dictionary keys.
  - Updated getJVMVersion to work with JDK 9+.
  - Added support for pickling of Java objects using optional module `jpype.pickle`
  - Fixed incorrect string conversion on exceptions. *str()* was incorrectly returning *getMessage* rather than *toString*.
  - Fixed an issue with JDK 12 regarding calling methods with reflection.
  - Removed limitations having to do with CallerSensitive methods. Methods affected are listed in caller_sensitive. Caller sensitive methods now receive an internal JPype class as the caller
  - Fixed segfault when converting null elements while accessing a slice from a Java object array.
  - PyJPMethod now supports the FunctionType API.
  - Tab completion with Jedi is supported. Jedi is the engine behind tab completion in many popular editors and shells such as IPython. Jedi version 0.14.1 is required for tab completion as earlier versions did not support annotations on compiled classes. Tab completion with older versions requires use of the IPython greedy method.
  - JProxy objects now are returned from Java as the Python objects that originate from. Older style proxy classes return the inst or dict. New style return the proxy class instance. Thus proxy classes can be stored on generic Java containers and retrieved as Python objects.

- **0.7.0 - 2019**

- Doc strings are generated for classes and methods.
- Complete rewrite of the core module code to deal unattached threads, improved hardening, and member management. Massive number of internal bugs were identified during the rewrite and corrected. See the ChangeLog-0.7 for details of all changes.
- API breakage:
  * Java strings conversion behavior has changed. The previous behavior was switchable, but only the default convert to Python was working. Converting to automatically lead to problems in which is was impossible to work with classes like StringBuilder in Java. To convert a Java string use `str()`. Therefore, string conversion is currently selected by a switch at the start of the JVM. The default shall be False starting in JPype 0.8. New code is encouraged to use the future default of False. For the transition period the default will be True with a warning if not policy was selected to encourage developers to pick the string conversion policy that best applies to their application.
  * Java exceptions are now derived from Python exception. The old wrapper types have been removed. Catch the exception with the actual Java exception type rather than `JException`.
  * Undocumented exceptions issued from within JPype have been mapped to the corresponding Python exception types such as `TypeError` and `ValueError` appropriately. Code catching exceptions from previous versions should be checked to make sure all exception paths are being handled.
  * Undocumented property import of Java bean pattern get/set accessors was removed as the default. It is available with `import jpype.beans`, but its use is discouraged.
- API rework:
  * JPype factory methods now act as base classes for dynamic class trees.
  * Static fields and methods are now available in object instances.
  * Inner classes are now imported with the parent class.
  * `jpype.imports` works with Python 2.7.
  * Proxies and customizers now use decorators rather than exposing internal classes. Existing `JProxy` code still works.
  * Decorator style proxies use `@JImplements` and `@JOverload` to create proxies from regular classes.
  * Decorator style customizers use `@JImplementionFor`
  * Module `jpype.types` was introduced containing only the Java type wrappers. Use `from jpype.types import *` to pull in this subset of JPype.
- `synchronized` using the Python `with` statement now works for locking of Java objects.
- Previous bug in initialization of arrays from list has been corrected.
- Added extra verbiage to the to the raised exception when an overloaded method could not be matched. It now prints a list of all possible method signatures.
- The following is now DEPRECATED
  * `jpype.reflect.*` - All class information is available with `.class_`
  * Unncessary `JException` from string now issues a warning.
- The followind is now REMOVED
  * Python thread option for `JPypeReferenceQueue`. References are always handled with with the Java cleanup routine. The undocumented `setUsePythonThreadForDaemon()` has been removed.

    - Undocumented switch to change strings from automatic to manual conversion has been removed.
    - Artifical base classes `JavaClass` and `JavaObject` have been removed.
    - Undocumented old style customizers have been removed.
    - Many internal jpype symbols have been removed from the namespace to prevent leakage of symbols on imports.
  - promoted *'–install-option'* to a *'–global-option'* as it applies to the build as well as install.
  - Added *'–enable-tracing'* to setup.py to allow for compiling with tracing for debugging.
  - Ant is required to build jpype from source, use `--ant=` with setup.py to direct to a specific ant.

- **0.6.3 - 2018-04-03**
  - Java reference counting has been converted to use JNI PushLocalFrame/PopLocalFrame. Several resource leaks were removed.
  - `java.lang.Class<>.forName()` will now return the java.lang.Class. Work arounds for requiring the class loader are no longer needed. Customizers now support customization of static members.
  - Support of `java.lang.Class<>`
    - `java.lang.Object().getClass()` on Java objects returns a java.lang.Class rather than the Python class
    - `java.lang.Object().__class__` on Java objects returns the python class as do all python objects
    - `java.lang.Object.class_` maps to the java statement `java.lang.Object.class` and returns the `java.lang.Class<java.lang.Object>`
    - java.lang.Class supports reflection methods
    - private fields and methods can be accessed via reflection
    - annotations are avaiable via reflection
  - Java objects and arrays will not accept setattr unless the attribute corresponds to a java method or field whith the exception of private attributes that begin with underscore.
  - Added support for automatic conversion of boxed types.
    - Boxed types automatically convert to python primitives.
    - Boxed types automatically convert to java primitives when resolving functions.
    - Functions taking boxed or primitives still resolve based on closest match.
  - Python integer primitives will implicitly match java float and double as per Java specification.
  - Added support for try with resources for `java.lang.Closeable`. Use python "with MyJavaResource() as resource:" statement to automatically close a resource at the end of a block.

- **0.6.2 - 2017-01-13**
  - Fix JVM location for OSX.
  - Fix a method overload bug.
  - Add support for synthetic methods

- **0.6.1 - 2015-08-05**
  - Fix proxy with arguments issue.
  - Fix Python 3 support for Windows failing to import winreg.

  - Fix non matching overloads on iterating java collections.

- **0.6.0 - 2015-04-13**
  - Python3 support.
  - Fix OutOfMemoryError.
- **0.5.7 - 2014-10-29**
  - No JDK/JRE is required to build anymore due to provided jni.h. To override this, one needs to set a JAVA_HOME pointing to a JDK during setup.
  - Better support for various platforms and compilers (MinGW, Cygwin, Windows)
- **0.5.6 - 2014-09-27**
  - *Note*: In this release we returned to the three point number versioning scheme.
  - Fix #63: 'property' object has no attribute 'isBeanMutator'
  - Fix #70: python setup.py develop does now work as expected
  - Fix #79, Fix #85: missing declaration of 'uint'
  - Fix #80: opt out NumPy code dependency by '–disable-numpy' parameter to setup. To opt out with pip append –install-option="–disable-numpy".
  - Use JVMFinder method of @tcalmant to locate a Java runtime
- **0.5.5.4 - 2014-08-12**
  - Fix: compile issue, if numpy is not available (NPY_BOOL n/a). Closes #77
- **0.5.5.3 - 2014-08-11**
  - Optional support for NumPy arrays in handling of Java arrays. Both set and get slice operators are supported. Speed improvement of factor 10 for setting and factor 6 for getting. The returned arrays are typed with the matching NumPy type.
  - Fix: add missing wrapper type 'JShort'
  - Fix: Conversion check for unsigned types did not work in array setters (tautological compare)
- **0.5.5.2 - 2014-04-29**
  - Fix: array setter memory leak (ISSUE: #64)
- **0.5.5.1 - 2014-04-11**
  - Fix: setup.py now runs under MacOSX with Python 2.6 (referred to missing subprocess function)
- **0.5.5 - 2014-04-11**
  - *Note* that this release is *not* compatible with Python 2.5 anymore!
  - Added AHL changes
    * replaced Python set type usage with new 2.6.x and higher
    * fixed broken Python slicing semantics on JArray objects
    * fixed a memory leak in the JVM when passing Python lists to JArray constructors
    * prevent ctrl+c seg faulting
    * corrected new[]/delete pairs to stop valgrind complaining
    * ship basic PyMemoryView implementation (based on numpy's) for Python 2.6 compatibility

  - Fast sliced access for primitive datatype arrays (factor of 10)
  - Use setter for Java bean property assignment even if not having a getter by @baztian
  - Fix public methods not being accessible if a Java bean property with the same name exists by @baztian (*Warning*: In rare cases this change is incompatibile to previous releases. If you are accessing a bean property without using the get/set method and the bean has a public method with the property's name you have to change the code to use the get/set methods.)
  - Make jpype.JException catch exceptions from subclasses by @baztian
  - Make more complex overloaded Java methods accessible (fixes https://sourceforge.net/p/jpype/bugs/69/) by @baztian and anonymous
  - Some minor improvements inferring unnecessary copies in extension code
  - Some JNI cleanups related to memory
  - Fix memory leak in array setters
  - Fix memory leak in typemanager
  - Add userguide from sourceforge project by @baztian
- **0.5.4.5 - 2013-08-25**
  - Added support for OSX 10.9 Mavericks by @rmangino (#16)
- **0.5.4.4 - 2013-08-10**
  - Rewritten Java Home directory Search by @marsam (#13, #12 and #7)
  - Stylistic cleanups of setup.py
- **0.5.4.3 - 2013-07-27**
  - Initial pypi release with most fixes for easier installation

1.7 Developer Guide

1.7.1 Overview

This document describes the guts of jpype. It is intended lay out the architecture of the jpype code to aid intrepid lurkers to develop and debug the jpype code once I am run over by a bus. For most of this document I will use the royal we, except where I am giving personal opinions expressed only by yours truly, the author Thrameos.

History

When I started work on this project it had already existed for over 10 years. The original developer had intended a much larger design with modules to support multiple languages such as Ruby. As such it was constructed with three layers of abstraction. It has a wrapper layer over Java in C++, a wrapper layer for the Python api in C++, and an abstraction layer intended to bridge Python and other interpreted languages. This multilayer abstraction ment that every debugging call had to drop through all of those layers. Memory management was split into multiple pieces with Java controlling a portion of it, C++ holding a bunch of resources, Python holding additional resources, and HostRef controlling the lifetime of objects shared between the layers. It also had its own reference counting system for handing Java references on a local scale.

This level of complexity was just about enough to scare off all but the most hardened programmer. Thus I set out to eliminate as much of this as I could. Java already has its own local referencing system to deal in the form of Local-Frames. It was simply a matter of setting up a C++ object to hold the scope of the frames to eliminate that layer. The

Java abstraction was laid out in a fashion somewhat orthagonally to the Java inheritance diagram. Thus that was reworked to something more in line which could be safely completed without disturbing other layers. The multilanguage abstraction layer was already pierced in multiple ways for speed. However, as the abastraction interwove throughout all the library it was a terrible lift to remove and thus required gutting the Python layer as well to support the operations that were being performed by the HostRef.

The remaining codebase is fairly slim and reasonably streamlined. This rework cut out about 30% of the existing code and sped up the internal operations. The Java C++ interface matches the Java class hierachy.

Architecture

JPype is split into several distinct pieces.

`jpype` Python module The majority of the front end logic for the toolkit is in Python jpype module. This module deals with the construction of class wrappers and control functions. The classes in the layer are all prefixed by `J`.

`_jpype` CPython module The native module is supported by a CPython module called `_jpype`. The `_jpype` module is located in `native/python` and has C style classes with a prefix `PyJP`.

This CPython layer acts as a front end for passing to the C++ layer. It performs some error checking. In addition to the module functions in `_JModule`, the module has multiple Python classes to support the native jpype code such as `_JClass`, `_JArray`, `_JValue`, `_JValue`, etc.

CPython API wrapper In addition to the exposed Python module layer, there is also a C++ wrapper for the Python API. This is located in `native/python` and has the prefix `JPPy` for all classes. `jp_pythontypes` wraps the required parts of the CPython API in C++ for use in the C++ layer.

C++ JNI layer The guts that drive Java are in the C++ layer located in `native/common`. This layer has the namespace `JP`. The code is divided into wrappers for each Java type, a typemanager for mapping from Java names to class instances, support classes for proxies, and a thin JNI layer used to help ensure rigerous use of the same design patterns in the code. The primary responsibility of this layer is type conversion and matching of method overloads.

Java layer In addition to the C++ layer, jpype has a native Java layer. This code is compiled as a "thunk" which is loaded into the JVM in the form of a a binary stored as a string. Code for Java is found in `native/java`. The Java layer is divided into two parts, a bootstrap loader and a jar containing the support classes. The Java layer is responsible managing the lifetime of shared Python, Java, and C++ objects.

1.7.2 `jpype` module

The `jpype` module itself is made of a series of support classes which act as factories for the individual wrappers that are created to mirror each Java class. Because it is not possible to wrap all Java classes with staticly created wrappers, instead jpype dynamically creates Python wrappers as requested by the user.

The wrapping process is triggered in two ways. The user can manually request creating a class by importing a class wrapper with jpype.imports or `JPackage` or by manually invoking it with `JClass`. Or the class wrapper can be created automatically as a result of a return type or exception thrown to the user.

Because the classes are created dynamically, the class structure uses a lot of Python meta programming. Each class wrapper derives from the class wrappers of each of the wrappers corresponding to the Java classes that each class extends and implements. The key to this is to hacked `mro`. The `mro` orders each of the classes in the tree such that the most drived class methods are exposed, followed by each parent class. This must be ordered to break ties resulting from multiple inheritance of interfaces. The factory classes are grafted into the type system using `__instancecheck__` and `__subtypecheck__`.

resource types

JPype largely maps to the same concepts as Python with a few special elements. The key concept is that of a Factory which serves to create Java resources dynamically as requested. For example there is no Python notation to create a `int[][]` as the concept of dimensions are fluid in Python. Thus a factory type creates the actual object instance type with `JArray(JInt,2)` Like Python objects, Java objects derives from a type object which is called `JClass` that serves as a meta type for all Java derived resources. Additional type like object `JArray` and `JInterface` serve to probe the relationships between types. Java object instances are created by calling the Java class wrapper just like a normal Python class. A number of pseudo classes serve as placeholders for Java types so that it is not necessary to create the type instance when using. These aliased classes are `JObject`, `JString`, and `JException`. Underlying all Java instances is the concept of a `jvalue`.

jvalue

In the earlier design, wrappers, primitives and objects were all seperate concepts. At the JNI layer these are unified by a common element called jvalue. A `jvalue` is a union of all primitives with the jobject. The jobject can represent anything derived from Java object including the pseudo class jstring.

This has been replaced with a Java slot concept which holds an instance of `JPValue` which holds a pointer to the C++ Java type wrapper and a Java jvalue union. We will discuss this object further in the CPython section.

Bootstrapping

The most challenging part in working with the jpype module other than the need to support both major Python versions with the same codebase is the bootstrapping of resources. In order to get the system working, we must pass the Python resources so the `_jpype` CPython module can acquire resources and then construct the wrappers for `java.lang.Object` and `java.lang.Class`. The key difficulty is that we need reflection to get methods from Java and those are part of `java.lang.Class`, but class inherits from `java.lang.Object`. Thus Object and the interfaces that Class inherits must all be created blindly. The order of bootstrapping is controlled by specific sequence of boot actions after the JVM is started in `startJVM`. The class instance `class_` may not be accessed until after all of the basic class, object, and exception types have been loaded.

Factories

The key objects exposed to the user (`JClass`, `JObject`, and `JArray`) are each factory meta classes. These classes serve as the gate keepers to creating the meta classes or object instances. These factories inherit from the Java class meta and have a `class_` instance inserted after the the JVM is started. They do not have exposed methods as they are shadows for action for actual Java types.

The user calls with the specified arguments to create a resource. The factory calls the `__new__` method when creating an instance of the derived object. And the C++ wrapper calls the method with internally construct resource such as `_JClass` or `_JValue`. Most of the internal calls currently create the resource directly without calling the factories. The gateway for this is `PyJPValue_create` which delegates the process to the corresponding specialized type.

Style

One of the aspects of the jpype design is elegance of the factory patterns. Rather than expose the user a large number of distinct concepts with different names, the factories provide powerfull functionality with the same syntax for related things. Boxing a primitive, casting to a specific type, and creating a new object are all tied together in one factory, `JObject`. By also making that factory an effective base class, we allow it to be used for `issubtype` and `isinstance`.

This philosophy is further enhanced by silent customizers which integrate Python functionality into the wrappers such that Java classes can be used effectively with Python syntax. Consistent use and misuse of Python concepts such as `with` for defining blocks such as try with resources and synchronized hide the underlying complexity and give the feeling to the user that the module is integrated completely as a solution such as jython.

When adding a new feature to the Python layer, consider carefully if the feature needs to be exposed a new function or if it can be hidden in the normal Python syntax.

JPype does somewhat break the Python naming conventions. Because Java and Python have very different naming schemes, at least part of the kit would have a different convention. To avoid having one portion break Python conventions and another part conform, we choose to use Java notation consistently throughout. Package names should be lower with underscores, classes should camel case starting upper, functions and method should be camel case starting lower. All private methods and classes start with a leading underscore and are not exported.

Customizers

There was a major change in the way the customizers work between versions. The previous system was undocumented and has now been removed, but as someone may have used of it previously, we will contrast it with the revised system so that the customizers can be converted.

In the previous system, a global list stored all customizers. When a class was created, it went though the list and asked the class if it matched that class name. If it matched, it altered the dict of members to be created so when the dynamic class was finished it had the custome behavior. This system wasn't very scalable as each customizer added more work to the class construction process.

The revised system works by storing a dictionary keyed to the class name. Thus the customizer only applies to the specific class targeted to the customizer. The customizer is specified using annotation of a prototype class making methods automatically copy onto the class. However, sometimes a customizer needs to be applied to an entire tree of classes such as all classes that implement `java.util.List`. To handle this case, the class creation system looks for a special method `__java_init__` in the tree of base classes and calls it on the newly created class. Most of the time the customization was the same simple pattern so we added a `sticky` flag to build the initialization method directly. This method can alter the class to make it add the new behavior. Note the word alter. Where before we changed the member prior to creating the class, here we are altering the class. Thus the customizer is expected to monkey patch the existing class. There is only one pattern of monkey patching that works on both Python 2 and Python 3 so be sure to use the `type.__setattr__` method of altering the class dictionary.

It is possible to apply customizers after the class has already been created because we operate by monkey patching. But there is a limitation that there can only be one `__java_init__` method and thus two customizers specifying a global behavior on the same class wrapper will lead to unexpected behavior.

1.7.3 `_jpype` CPython module

Diving deeper into the onion, we have the Python front end. This is divided into a number of distinct pieces. Each piece is found under `native/python` and is named according to the piece it provides. For example, `PyJPModule` is found in the file `native/python/pyjp_module.cpp`

Earlier versions of the module had all of the functionality in the modules global space. This functionality is now split into a number of classes. These classes each have a constructor that is used to create an instance which will correspond to a Java resource such as class, array, method, or value.

Jpype objects work with the inner layers by inheriting from a set of special `_jpype` classes. This class hiarachy is mantained by the meta class `_jpype._JClass`. The meta class does type hacking of the Python API to insert a reserved memory slot for the `JPValue` structure. The meta class is used to define the Java base classes:

- `_JClass` - Meta class for all Java types which maps to a java.lang.Class extending Python type.
- `_JArray` - Base class for all Java array instances.

- `_JObject` - Base type of all Java object instances extending Python object.
- `_JNumberLong` - Base type for integer style types extending Python int.
- `_JNumberFloat` - Base type for float style types extending Python float.
- `_JNumberChar` - Special wrapper type for JChar and java.lang.Character types extending Python float.
- `_JException` - Base type for exceptions extending Python Exception.
- `_JValue` - Generic capsule representing any Java type or instance.

These types are exposed to Python to implement Python functionality specific to the behavior expected by the Python type. Under the hood these types are largely ignored. Instead the internal calls for the Java slot to determine how to handle the type. Therefore, internally often Python methods will be applied to the "wrong" type as the requirement for the method can be satisfied by any object with a Java slot rather than a specific type.

See the section regarding Java slots for details.

`PyJPModule` module

This is the front end for all the global functions required to support the Python native portion. Most of the functions provided in the module are for control and auditing.

Resources are created by setting attributes on the `_jpype` module prior to calling `startJVM`. When the JVM is started each of th required resources are copied from the module attribute lists to the module internals. Setting the attributes after the JVM is started has no effect. Resources are verified to exist when the JVM is started and any missing resource are reported as an error.

`_JClass` class

The class wrappers have a metaclass `_jpyep._JClass` which serves as the guardian to ensure the slot is attached, provide for the inheritance checks, and control access to static fields and methods. The slot holds a java.lang.Class instance but it does not have any of the methods normally associate with a Java class instance exposed. A java.lang.Class instance can be converted to a Jave class wrapper using `JClass`.

`_JMethod` class

This class acts as descriptor with a call method. As a descriptor accessing its methods through the class will trigger its `__get__` function, thus getting ahold of it within Python is a bit tricky. The `__get__` mathod is used to bind the static unbound method to a particular object instance so that we can call with the first argument as the `this` pointer.

It has some reflection and diagnostics methods that can be useful it tracing down errors. The beans methods are there just to support the old properties API.

The naming on this class is a bit deceptive. It does not correspond to a single method but rather all the overloads with the same name. When called it passes to with the arguments to the C++ layer where it must be resolved to a specific overload.

This class is stored directly in the class wrappers.

`_JField` class

This class is a descriptor with `__get__` and `__set__` methods. When called at the static class layer it operates on static fields. When called on a Python object, it binds to the object making a `this` pointer. If the field is static, it will continue to access the static field, otherwise, it will provide access to the member field. This trickery allows both static and member fields to wrap as one type.

This class is stored directly in the class wrappers.

`_JArray` class

Java arrays are extensions of the Java object type. It has both methods associated with java.lang.Object and Python array functionality. Primitives have specialized implementations to allow for the Python buffer API.

`_JMonitor` class

This class provides `synchronized` to JPype. Instances of this class are created and held using `with`. It has two methods `__enter__` and `__exit__` which hook into the Python RAII system.

`_JValue` class

Java primitive and object instance derive from special Python derived types. These each have the Python functionality to be exposed and a Java slot. The most generic of these is `_JValue` which is simply a capsule holding the Java C++ type wrapper and a Java jvalue union. CPython methods for the `PyJPValue` apply to all CPython objects that hold a Java slot.

Specific implementation exist for object, numbers, characters, and exceptions. But fundimentally all are treated the same internally and thus the CPython type is effectively erased outside of Python.

Unlike `jvalue` we hold the object type in the C++ `JPValue` object. The class reference is used to determine how to match the arguments to methods. The class may not correspond to the actual class of the object. Using a class other than the actual class serves to allow an object to be cast and thus treated like another type for the purposes of overloading. This mechanism is what allows the `JObject` factory to perform a typecast to make an object instance act like one of its base classes..

1.7.4 Java Slots

THe key to achieving reasonable speed within CPython is the use of slots. A slot is a dedicated memory location that can be accessed without consulting the dictionary or bases of an object. CPython achieve this by reserving space within the type structure and by using a set of bit flags so that it can avoid costly. The reserved space in order by number and thus avoids the need to access the dictionary while the bit flags serve to determine the type without traversing the `__mro__` structure. We had to implement the same effect which deriving from a wide variety for Python types including type, object, int, long, and Exception. Adding the slot directly to the type and objects base memory does not work because these types all have different memory layouts. We could have a table look up based on the type but because we must obey both the CPython and the Java object hierarchy at the same time it cannot be done within the memory layout of Python objects. Instead we have to think outside the box, or rather outside the memory footprint of Python objects.

CPython faces the same conflict internally as inheritance often forces adding a dictionary or weak reference list onto a variably size type sych as long. For those cases it adds extract space to the basesize of the object and then ignores that space for the purposes of checking inheritance. It pairs this with an offset slot that allows for location of the dynamic placed slots. We cannot replicate this in the same way because the CPython interals are all specialize static members and there is no provision for introducting user defined dynamic slots.

Therefore, instead we will add extra memory outside the view of Python objects though the use of a custom allocator. We intercept the call to create an object allocation and then call the regular Python allocators with the extra memory added to the request. As our extrs slot has resource in the form of Java global references associated with it, we must deallocate those resource regardless of the type that has been extended. We perform this task by creating a custom finalize method to serve as the destructor. Thus a Java slot requires overriding each of `tp_alloc`, `tp_free` and

`tp_finalize`. The class meta gatekeeper creates each type and verifies that the required hooks are all in place. If the user tries to bypass this it should produce an error.

In place of Python bit flags to check for the presence of a Java slot we instead test the slot table to see if our hooks are in place. We can test if the slot is present by looking to see if both *tp_alloc* and *tp_finalize* point to our Java slot handlers. This means we are still effectively a slot as we can test and access with O(1).

Accessing the slot requires testing if the slot exists for the object, then computing the sice of the object using the basesize and itemsize associate with the type and then offsetting the Python object pointer appropriately. The overall cost is O(1), though is slightly more heavy that directly accesssing an offset.

1.7.5 CPython API layer

To make creation of the C++ layer easier a thin wrapper over the CPython API was developed. This layer provided for handling the CPython referencing using a smart pointer, defines the exception handling for Python, and provides resource hooks for duck typing of the `_jpype` classes.

This layer is located with the rest of the Python codes in `native/python`, but has the prefix `JPPy` for its classes. As the bridge between Python and C++, these support classes appear in both the `_jpype` CPython module and the C++ JNI layer.

Exception handling

A key piece of the jpype interaction is the transfer of exceptions from Java to Python. To accomplish this Python method that can result in a call to Java must have a `try` block around the contents of the function.

We use a routine pattern of code to interact with Java to achieve this:

```
PyObject* dosomething(PyObject* self, PyObject* args)
{
   // Tell the logger where we are
   JP_PY_TRY("dosomething");

   // Make sure there is a jvm to receive the call.
   ASSERT_JVM_RUNNING("dosomething");

   // Make a resource to capture any Java local references
   JPJavaFrame frame;

   // Call our Java methods
   ...

   // Return control to Python
   return obj.keep();

   // Use the standard catch to transfer any exceptions back
   // to Python
   JP_PY_CATCH(NULL);
}
```

All entry points from Python into `_jpype` should be guarded with this pattern.

There are exceptions to this pattern such as removing the logging, operating on a call that does not need the JVM running, or operating where the frame is already supported by the method being called.

Python referencing

One of the most miserable aspects of programming with CPython is the relative inconsistancy of referencing. Each method in Python may use a Python object or steal it, or it may return a borrowed reference or give a fresh reference. Similar command such as getting an element from a list and getting an element from a tuple can have different rules. This was a constant source of bugs requiring consultation of the Python manual for every line of code. Thus we wrapped all of the Python calls we were required to work with in `jp_pythontypes`.

Included in this wrapper is a Python reference counter called `JPPyObject`. Whenever an object is returned from Python it is immediately placed in smart pointer `JPPyObject` with the policy that it was created with such as `use_`, `borrowed_`, `claim_` or `call_`.

`use_` This policy means that the reference counter needs to be incremented and the start and the end. We must reference it because if we don't and some Python call destroys the refernce out from under us, the system may crash and burn.

`borrowed_` This policy means we were to be give a borrowed reference that we are expected to reference and unreference when complete, but the command that returned it can fail. Thus before reference it, the system must check if an error has occurred. If there is an error, it is promoted to an exception.

`claim_` This policy is used when we are given a new object with is already referenced for us. Thus we are to steal the reference for the duration of our use and then dereference when we are done to keep it from leaking.

`call_` This policy both steals the reference and verifies there were no errors prior to continuing. Errors are promoted to exceptions when this reference is created.

If we need to pass an object which is held in a smart pointer to Python which requires a reference, we call keep on the reference which transfers control to a `PyObject*` and prevents the pointer from removing the reference. As the object handle is leaving our control keep should only be called the return statement. The smart pointer is not used on method passing in which the parent explicitly holds a reference to the Python object. As all tuples passed as arguments operate like this, that means much of the API accepts bare `PyObject*` as arguments. It is the job of the caller to hold the reference for its scope.

On CPython extensions

CPython is somewhat of a nightmare to program in. It is not that they did not try to document the API, but it is darn complex. The problems extend well beyond the reference counting system that we have worked around. In particular, the object model though well developed is very complex, often to get it to work you must follow letter for letter the example on the CPython user guide, and even then it may all go into the ditch.

The key problem is that there are a lot of very bad examples of how to write CPython extension modules out there. Often the these examples bypass the appropriate macro and just call the field, or skip the virtual table and try to call the Python method directly. It is true that these things do not break there example, but they are conditioned on these methods they are calling directly to be the right one for the job, but depends a lot on what the behavior of the object is supposed to be. Get it wrong and you get really nasty segfault.

CPython itself may be partly responsible for some of these problems. They generally seem to trust the user and thus don't verify if the call makes sense. It is true that it will cost a little speed to be aggressive about checking the type flags and the allocator match, but not checking when the error happens, means that it fails far from the original problem source. I would hope that we have moved beyond the philosophy that the user should just to whatever they want so it runs as fast as possible, but that never appears to be the case. Of course, I am just opining from the outside of the tent and I am sure the issues are much more complicated it appears superficially. Then again if I can manage to provide a safe workspace while juggling the issues of multiple virtual machines, I am free to have opinions on the value of trading performance and safety.

In short when working on the extension code, make sure you do everything by the book, and check that book twice. Always go through the types virtual table and use the propery macros to access the resources. Miss one line in some complex pattern even once and you are in for a world of hurt. There are very few guard rails in the CPython code.

1.7.6 C++ JNI layer

The C++ layer has a number of tasks. It is used to load thunks, call JNI methods, provide reflection of classes, determine if a conversion is possible, perform conversion, match arguments to overloads, and convert return values back to Java.

Memory management

Java provides built in memory management for controlling the lifespan of Java objects that are passed through JNI. When a Java object is created or returned from the JVM it returns a handle to object with a reference counter. To manage the lifespan of this reference counter a local frame is created. For the duration of this frame all local references will continue to exist. To extend the lifespan either a new global reference to the object needs to be created, or the object needs to be kept. When the local frame is destroyed all local references are destroyed with the exception of an optional specified local return reference.

We have wrapped the Java reference system with the wrapper `JPLocalFrame`. This wrapper has three functions. It acts as a RAII (Resource acquisition is initialization) for the local frame. Further, as creating a local frame requires creating a Java env reference and all JNI calls require access to the env, the local frame acts as the front end to call all JNI calls. Finally as getting ahold of the env requires that the thread be attached to Java, it also serves to automatically attach threads to the JVM. As accessing an unbound thread will cause a segmentation fault in JNI, we are now safe from any threads created from within Python even those created outside our knowledge. (I am looking at you spyder)

Using this pattern makes the JPype core safe by design. Forcing JNI calles to be called using the frame ensures:

- Every local reference is destroyed.
- Every thread is properly attached before JNI is used.
- The pattern of keep only one local reference is obeyed.

To use a local frame, use the pattern shown in this example.

```
jobject doSomeThing(std::string args)
{
    // Create a frame at the top of the scope
    JPLocalFrame frame;

    // Do the required work
    jobject obj =frame.CallObjectMethodA(globalObj, methodRef, params);

    // Tell the frame to return the reference to the outer scope.
    //   once keep is called the frame is destroyed and any
    //   call will fail.
    return frame.keep(obj);
}
```

Note that the value of the object returned and the object in the function will not be the same. The returned reference is owned by the enclosing local frame and points to the same object. But as its lifespan belongs to the outer frame, its location in memory is different. You are allowed to `keep` a reference that was global or was passed in, in either of those case, the outer scope will get a new local reference that points to the same object. Thus you don't need to track the origin of the object.

The changing of the value while pointing is another common problem. A routine error is to get a local reference, call `NewGlobalRef` and then keeping the local reference rather than the shiny new global reference it made. This is not like the Python reference system where you have the object that you can ref and unref. Thus make sure you always store only the global reference.

```
jobject global;

// we are getting a reference, may be local, may be global.
// either way it is borrowed and it doesn't belong to us.
void elseWhere(jvalue value)
{
  JPLocalFrame frame;

  // Bunch of code leading us to decide we need to
  // hold the resource longer.
  if (cond)
  {
    // okay we need to keep this reference, so make a
    // new global reference to it.
    global = frame.NewGlobalRef(value.l);
  }
}
```

But don't mistake this as an invitation to make global references everywhere. Global reference are global, thus will hold the member until the reference is destroyed. C++ exceptions can lead to missing the unreference, thus global references should only happen when you are placing the Java object into a class member variable or a global variable.

To help manage global references, we have `JPRef<>` which holds a global reference for the duration of the C++ lifespace. This is the base class for each of the global reference types we use.

```
typedef JPRef<jclass> JPClassRef;
typedef JPRef<jobject> JPObjectRef;
typedef JPRef<jarray> JPArrayRef;
typedef JPRef<jthrowable> JPThrowableRef;
```

For functions that expect the outer scope to already have created a frame for this context, we use the pattern of extending the outer scope rather than creating a new one.

```
jobject doSomeThing(JPLocalFrame& frame, std::string args)
{
    // Do the required work
    jobject obj = frame.CallObjectMethodA(globalObj, methodRef, params);

    // We must not call keep here or we will terminate
    // a frame we do not own.
    return obj;
}
```

Although the system we have set up is "safe by design", there are things that can go wrong is misused. If the caller fails to create a frame prior to calling a function that returns a local reference, the reference will go into the program scoped local references and thus leak. Thus, it is usually best to force the user to make a scope with the frame extension pattern. Second, if any JNI references that are not kept or converted to global, it becomes invalid. Further, since JNI recycles the reference pointer fairly quickly, it most likely will be pointed to another object whose type may not be expected. Thus, best case is using the stale reference will crash and burn. Worse case, the reference will be a live reference to another object and it will produce an error which seems completely irrelevant to anything that was being called. Horrible case, the live object does not object to bad call and it all silently proceeds down the road another two miles before coming to flaming death.

Moral of the story, always create a local frame even if you are handling a global reference. If passed or returned a reference of any kind, it is a borrowed reference belonging to the caller or being held by the current local frame. Thus it must be treated accordingly. If you have to hold a global use the appropraite `JPRef` class to ensure it is exception and dtor safe. For further information read `native/common/jp_javaframe.h`.

Type wrappers

Each Java type has a C++ wrapper class. These classes provide a number of methods. Primitives each have their own unit type wrapper. Object, arrays, and class instances share a C++ wrapper type. Special instances are used for `java.lang.Object` and `java.lang.Class`. The type wrapper are named for the class they wrap such as `JPIntType`.

Type conversion

For type conversion, a C++ class wrapper provides four methods.

`canConvertToJava` This method must consult the supplied Python object to determine the type and then make a determination of whether a conversion is possible. It reports `none_` if there is no possible conversion, `explicit_` if the conversion is only acceptable if forced such as returning from a proxy, `implicit_` if the conversion is possible and acceptable as part of an method call, or `exact_` if this type converts without ambiguity. It is excepted to check for something that is already a Java resource of the correct type such as `JPValue`, or something this is implementing the behavior as an interface in the form of a `JPProxy`.

`convertToJava` This method consults the type and produces a conversion. The order of the match should be identical to the `canConvertToJava`. It should also handle values and proxies.

`convertToPythonObject` This method takes a jvalue union and converts it to the corresponding Python wrapper instance.

`getValueFromObject` This converts a Java object into a `JPValue` corresponding. This unboxes primitives.

Array conversion

In addition to converting single objects, the type rewrappers also serve as the gateway to working with arrays of the specified type. Five methods are used to work with arrays: `newArrayInstance`, `getArrayRange`, `setArrayRange`, `getArrayItem`, and `setArrayItem`.

Invocation and Fields

To convert a return type produced from a Java call, each type needs to be able to invoke a method with that return type. This corresponses the underlying JNI design. The methods invoke and invokeStatic are used for this purpose. Similarly accessing fields requires type conversion using the methods `getField` and `setField`.

Instance versus Type wrappers

Instances of individual Java classes are made from `JPClass`. However, two special sets of conversion rules are required. These are in the form of specializations `JPObjectBaseClass` and `JPClassBaseClass` corresponding to `java.lang.Object` and `java.lang.Class`.

Support classes

In addition to the type wrappers, there are several support classes. These are:

`JPTypeManager` The typemanager serves as a dict for all type wrappers created during the operation.

`JPReferenceQueue` Lifetime manager for Java and Python objects.

`JPProxy` Proxies implement a Java interface in Python.

JPClassLoader Loader for Java thunks.

JPEncoding Decodes and encodes Java UTF strings.

JPTypeManager

C++ typewrappers are created as needed. Instance of each of the primitives along with `java.lang.Object` and `java.lang.Class` are preloaded. Additional instances are created as requested for individual Java classes. Currently this is backed by a C++ map of string to class wrappers.

The typemanager provides a number lookup methods.

```
// Call from within Python
JPClass* JPTypeManager::findClass(const string& name)

// Call from a defined Java class
JPClass* JPTypeManager::findClass(jclass cls)

// Call used when returning an object from Java
JPClass* JPTypeManager::findClassForObject(jobject obj)
```

JPReferenceQueue

When a Python object is presented to Java as opposed to a Java object, the lifespan of the Python object must be extended to match the Java wrapper. The reference queue adds a reference to the Python object that will be removed by the Java layer when the garbage collection deletes the wrapper. This code is almost entirely in the Java library, thus only the portion to support Java native methods appears in the C++ layer.

Once started the reference queue is mostly transparent. registerRef is used to bind a Python object live span to a Java object.

```
void JPReferenceQueue::registerRef(jobject obj, PyObject* hostRef)
```

JPProxy

In order to call Python functions from within Java, a Java proxy is used. The majority of the code is in Java. The C++ code holds the Java native portion. The native implement of the proxy call is the only place in with the pattern for reflecting Python exceptions back into Java appears.

As all proxies are ties to Python references, this code is strongly tied to the reference queue.

JPClassLoader

This code is responsible for loading the Java class thunks. As it is difficult to ensure we can access a Java jar from within Python, all Java native code is stored in a binary thunk compiled into the C++ layer as a header. The class loader provides a way to load this embedded jar first by bootstrapping a custom Java classloader and then using that classloader to load the internal jar.

The classloader is mostly transparent. It provides one method called findClass which loads a class from the internal jar.

```
jclass JPClassLoader::findClass(string name)
```

JPEncoding

Java concept of UTF is pretty much out of sync with the rest of the world. Java used 16 bits for its native characters. But this was inadequate for all of the unicode characters, thus longer unicode character had to be encoded in the 16 bit space. Rather the directly providing methods to convert to a standard encoding such as UTF8, Java used UTF16 encoded in 8 bits which they dub Modified-UTF8. `JPEncoding` deals with converting this unusual encoding into something that Python can understand.

The key method in this module is transcribe with signature

```
std::string transcribe(const char* in, size_t len,
    const JPEncoding& sourceEncoding,
    const JPEncoding& targetEncoding)
```

There are two encodings provided, `JPEncodingUTF8` and `JPEncodingJavaUTF8`. By selecting the source and traget encoding transcribe can convert to or from Java to Python encoding.

Incidentally that same modified UTF coding is used in storing symbols in the class files. It seems like a really poor design choice given they have to document this modified UTF in multiple places. As far as I can tell the internal converter only appears on `java.io.DataInput` and `java.io.DataOutput`.

1.7.7 Java native code

At the lowest level of the onion is the native Java layer. Although this layer is most remote from Python, ironically it is the easiest layer to communicate with. As the point of jpype is to communicate with Java, it is possible to directly communicate with the jpype Java internals. These can be imported from the package `org.jpype`. The code for the Java layer is located in `native/java`. It is compiled into a jar in the build directory and then converted to a C++ header to be compiled into the `_jpype` module.

The Java layer currently houses the reference queue, a classloader which can load a Java class from a bytestream source, the proxy code for implementing Java interfaces, and a memory compiler module which allows Python to directly create a class from a string.

1.7.8 Tracing

Because the relations between the layers can be daunting especially when things go wrong. The CPython and C++ layer have a built in logger. This logger must be enabled with a compiler switch to activate. To active the logger, touch one of the cpp files in the native directory to mark the build as dirty, then compile the `jpype` module with:

```
python setup.py --enable-tracing devel
```

Once built run a short test program that demonstrates the problem and capture the output of the terminal to a file. This should allow the developer to isolate the fault to specific location where it failed.

To use the logger in a function start the `JP_TRACE_IN(function_name)` which will open a `try catch` block.

The JPype tracer can be augmented with the Python tracing module to give a very good picture of both JPype and Python states at the time of the crash. To use the Python tracing, start Python with...

```
python -m trace --trace myscript.py
```

1.7.9 Debugging issues

If the tracing function proves inadequate to identify a problem, we often need to turn to a general purpose tool like gdb or valgrind. The JPype core is not easy to debug. Python can be difficult to properly monitor especially with tools like valgrind due to its memory handling. Java is also challenging to debug. Put them together and you have the mother of all debugging issues. There are a number of complicating factors. Let us start with how to debug with gdb.

Gdb runs into two major issues, both tied to the signal handler. First, Java installs its own signal handlers that take over the entire process when a segfault occurs. This tends to cause very poor segfault stacktraces when examining a core file, which often is corrupt after the first user frame. Second, Java installs its signal handlers in such as way that attempting to run under a debugger like gdb will often immediately crash preventing one from catching the segfault before Java catches it. This makes for a catch 22, you can't capture a meaningful non-interactively produced core file, and you can't get an interactive session to work.

Fortunately there are solutions to the interactive session issue. By disabling the SIGSEGV handler, we can get past the initial failure and also we can catch the stack before it is altered by the JVM.

```
gdb -ex 'handle SIGSEGV nostop noprint pass' python
```

Thus far I have not found any good solutions to prevent the JVM from altering the stack frames when dumping the core. Thus interactive debugging appears to be the best option.

There are additional issues that one should be aware of. Open-JDK 1.8 has had a number of problems with the debugger. Starting JPype under gdb may trigger, may trigger the following error.

```
gdb.error: No type named nmethod.
```

There are supposed to be fixes for this problem, but none worked for me. Upgrading to Open-JDK 9 appears to fix the problem.

Another complexity with debugging memory problems is that Python tends to hide the problem with its allocation pools. Rather than allocating memory when a new object is request, it will often recycle and existing object which was collect earlier. The result is that an object which turns out is still live becomes recycled as a new object with a new type. Thus suddenly a method which was expected to produce some result instead vectors into the new type table, which may or may not send us into segfault land depending on whether the old and new objects have similar memory layouts.

This can be partially overcome by forcing Python to use a different memory allocation scheme. This can avoid the recycling which means we are more likely to catch the error, but at the same time means we will be excuting different code paths so we may not reach a similar state. If the core dump is vectoring off into code that just does not make sense it is likely caused by the memory pools. Starting Python 3, it is possible to select the memory allocation policy through an enviroment variable. See the `PYTHONMALLOC` setting for details.

1.7.10 Future directions

Although the majority of the code has been reworked for JPype 0.7, there is still further work to be done. Almost all Java constructs can be exercised from within Python, but Java and Python are not static. Thus, we are working on further improvements to the jpype core focusing on making the package faster, more efficient, and easier to maintain. This section will discuss a few of these options.

Java based code is much easier to debug as it is possible to swap the thunk code with an external jar. Further, Java has much easier management of resources. Thus pushing a portion of the C++ layer into the Java layer could further reduce the size of the code base. In particular, deciding the order of search for method overloads in C++ attempts to reconstruct the Java overload rules. But these same rules are already available in Java. Further, the C++ layer is designed to make many frequent small calls to Java methods. This is not the preferred method to operate in JNI. It is better to have specialized code in Java which preforms large tasks such as collecting all of the fields needed for a type

wrapper and passing it back in a single call, rather than call twenty different general purpose methods. This would also vastly reduce the number of `jmethods` that need to be bound in the C++ layer.

The world of JVMs is currently in flux. Jpype needs to be able to support other JVMs. In theory, so long a JVM provides a working JNI layer, there is no reason the jpype can't support it. But we need loading routines for these JVMs to be developed if there are differences in getting the JVM launched.

There is a project page on github shows what is being developed for the next release. Series 0.6 was usable, but early versions had notable issues with threading and internal memory management concepts had to be redone for stability. Series 0.7 is the first verion after rewrite for simplication and hardening. I consider 0.7 to be at the level of production quality code suitable for most usage though still missing some needed features. Series 0.8 will deal with higher levels of Python/Java integration such as Java class extension and pickle support. Series 0.9 will be dedicated to any additional hardening and edge cases in the core code as we should have complete integration. Assuming everything is completed, we will one day become a real boy and have a 1.0 release.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

j

Index

Symbols

A

D

G

I

J

R

S