Clamp - Seamless integration of Python and Java

Jim Baker, Rackspace

jim.baker@rackspace.com

Clamp background

- Part of the jythontools project (https://github.com/jythontools)
- Improve Python <=> Java integration (which is already very good)
- Enable precise layout of the generated Java bytecode for Python classes
- So they can be used as modern Java classes - annotation metadata, type signatures
- Jar packaging into site-packages
- Or entire Jython installation wrapped into a **single jar**

Benefits

- JVM frameworks can readily work with clamped code, oblivious of its source
- Especially need single jar support
- Developers can stay as much in Python as possible
- Working on SQLAlchemy-like DSL

Development

- Heavily uses support for metaprogramming in Python and Java
- If there's a metaprogramming facility, we seem to be either using it now or exploring it
- Useful outcome already: merge `type`, `java.lang.Class` so equivalent types for metaclass usage

Example: Storm

- "Real-time" complex event processing system
- Runs topology of storms, bolts to process events ("tuples")
- Can support at-least-once, exactly-once semantics

What is needed for Storm

- No-arg constructors
- Specify `serialVersionUid`
- Single jar support
- Needs to be able to resolve class names to classes (`Class.forName`)

Python class, extending Java interfaces


```
from java.io import Serializable
from java.util.concurrent import Callable

class BarClamp(Callable, Serializable):

  def call(self):
    return 42
```

How to use from Java?

- Can use as a Java callback
- Can use JSR-223 or embed Jython runtime into Java
- But cannot directly use as-is from Java
- Specifically no way to construct a new instance of the class
- Definitive Guide to Jython goes into detail: object factories, etc
- Good if you are already using JSR-223 etc, want to support scripting, etc
- Bad if you just want to use Python code in your framework as one component, instead of having to write the component in Java
- Note that solutions like Scala, Clojure, face similar issues: they need to describe how they should be exposed with a Java API
- Solution: Clamp!

Python class, clamped

Simpy add the Clamp base, a metaclass:

```
from java.io import Serializable
from java.util.concurrent import Callable
from clamp import clamp_base

BarBase = clamp_base("bar")  # Java package prefix

class BarClamp(BarBase, Callable, Serializable):

  def call(self):
    return 42
```

Clamping your class

Key insight: ahead-of-time builds through setuptools:

```
import ez_setup
ez_setup.use_setuptools()

from setuptools import setup, find_packages

setup(
  name = "clamped",
  version = "0.1",
  packages = find_packages(),
  install_requires = ["clamp>=0.3"],
  clamp = ["clamped"],
)
```

Using from Java

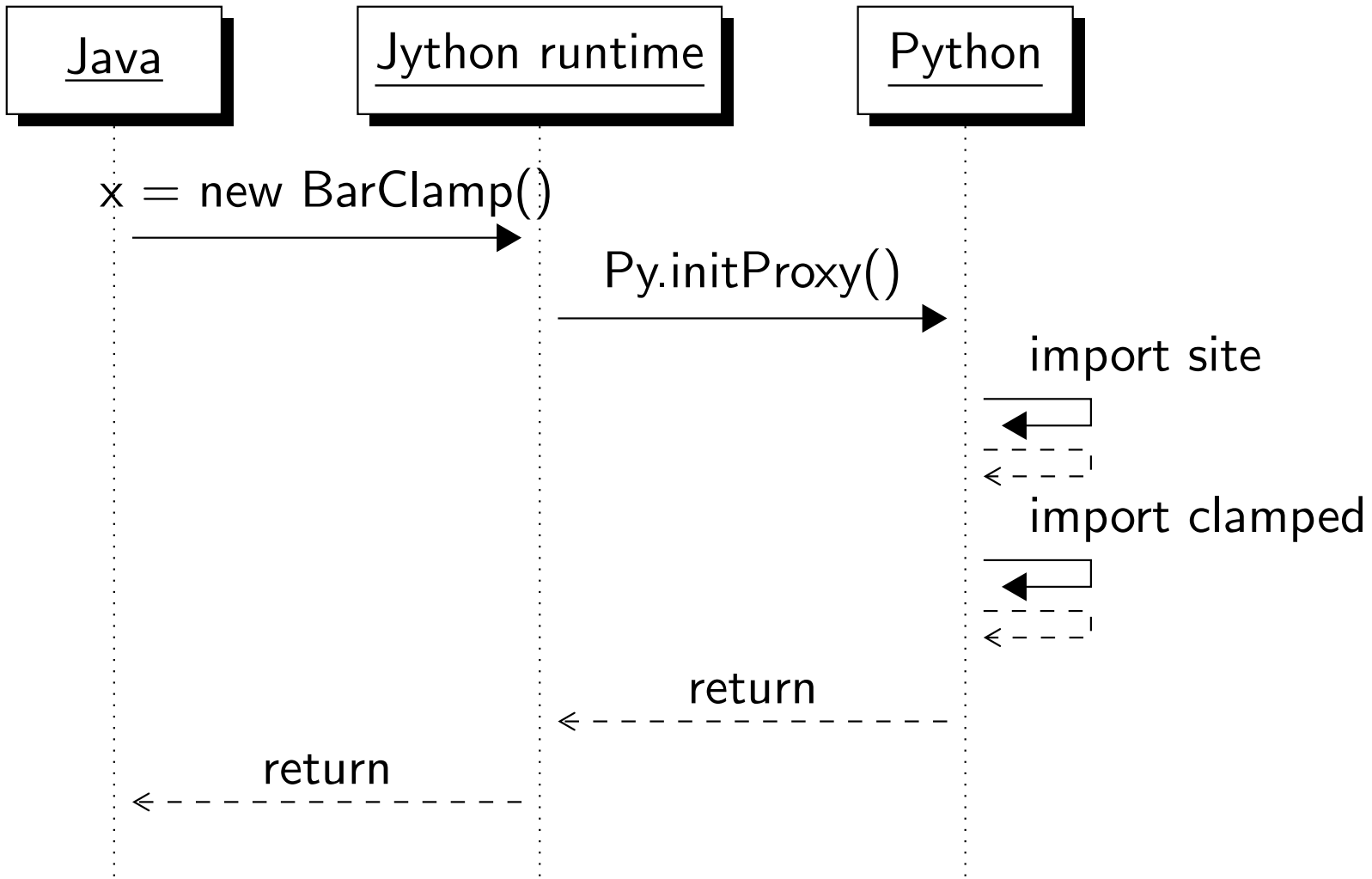
Simply import clamped Python classes into Java code!

```
import bar.clamped.BarClamp;

public class UseClamped {
  public static void main(String[] args) {
    BarClamp barclamp = new BarClamp();
    try {
      System.out.println("BarClamp: " +
        barclamp.call());
    } catch (Exception ex) {
      System.err.println("Exception: " + ex);
    }
  }
}
```

Java/Python rendezvous

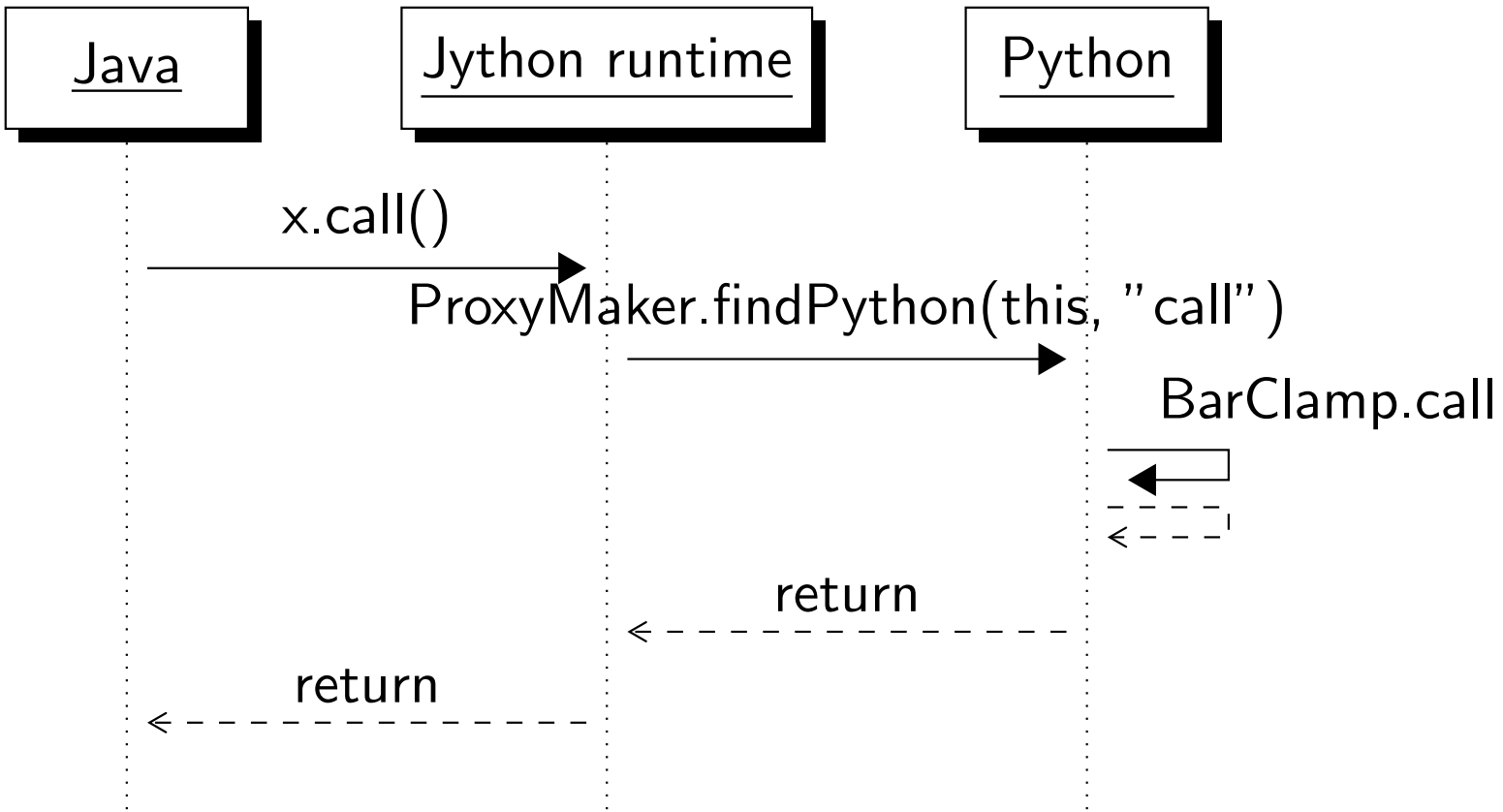
Initializing BarClamp



Loading Jython runtime

- When does the Jython runtime get loaded? (bootstrap problem!)
- `Py.initProxy` calls `Py.getThreadState`, which in turn loads the runtime as necessary!
- `ThreadState` extends `java.lang.ThreadLocal`

Making a call



Maintaining state

- `ThreadState` also maintains state through the call stack, from Java to Python and back, as necessary
- Other critical call: `findPython`, so that proxies can get the corresponding Python code!

Metaprogramming

- What do we mean by metaprogramming?
- Solution to every problem in CS is indirection...
- Except for indirection!

Better DSLs through metaclasses

- Declarative DSL
- Construct a mapper metaclass (similar to and inspired by SQLAlchemy)

API design aside

- Original clamp work was writing this in Java
- Getting too complex
- Let's use Python instead!
- Even if in places we are generating some Java bytecode!

Metaclass factory

Integrations can associate more info with metaclass base:

```
def clamp_base(package, proxy_maker=ClampProxyMaker):
  def _clamp_closure(package, proxy_maker):
    class ClampProxyMakerMeta(type):
      def __new__(cls, name, bases, d):
        d = dict(d)
        d['__proxymaker__'] = proxy_maker(
          package=package)
        return type.__new__(cls, name, bases, d)
    return ClampProxyMakerMeta
  class ClampBase(object):
    __metaclass__ = _clamp_closure(
      package=package, proxy_maker=proxy_maker)
  return ClampBase
```

Proxy maker

- Heart of Clamp!
- Need to **precisely** generate in Java bytecode a Java class that implements interfaces and/or extends a class
- With a given name, because Java can be finicky about that (`Class.forName`, any static linkage)
- Constructors (no-arg at least)
- Static fields such as `serialVersionUUID` to support `Serializable`
- Use the same bytecode!

`__proxymaker__` protocol

- Specific to Jython
- New `__proxymaker__` protocol allows for a `CustomMaker` to intercept the construction of a Java proxy
- Code generation
- Saving bytes
- Turning into a class via a `ClassLoader`

Code generation example: <clinit>

Every class has a specially-named hidden method, <clinit>, to support initialization for the class itself:

```
public static final long serialVersionUID;
static {
  serialVersionUID = 42L;
}
```

- <clinit> automatically emitted by the Java compiler
- **Clamp needs to do the same**

Code generation, in Python

Assuming self.constants is initialized like so:

```
{ "serialVersionUID":
  (java.lang.Long(42), java.lang.Long.TYPE), ... }
```

`serialVersionUID` and dynamic typing

- Q: What is the correct value of `serialVersionUID`?
- A: 1L - this constant is to support dynamic evolution, but dynamic typing does that for us anyway!

Code generation

Standard visitor approach:

```
def doConstants(self):
  code = self.classfile.addMethod("<clinit>",
    ProxyCodeHelpers.makeSig("V"), Modifier.STATIC)
  for constant, (value, constant_type) in sorted(
      self.constants.iteritems()):
    self.classfile.addField(
      constant,
      CodegenUtils.ci(constant_type),
      Modifier.PUBLIC | Modifier.STATIC |
      Modifier.FINAL)
    code.visitLdcInsn(value)
    code.putstatic(self.classfile.name,
      constant, CodegenUtils.ci(constant_type))
  code.return_()
```

Current DSL

http://clamp.readthedocs.org

Locating jars

- Works because of this importer in `sys.path`: `'__classpath__'`
- Other `sys.path` magic allows for dynamic addition to the `sys.path`
- How can we do this?
- Jython uses custom `ClassLoader` objects - Java is very flexible with respect to how to find, load classes

pth support

- Makes `sys.path` even more flexible
- site-package packages can be added to `sys.path`
- Implemented using `jar.pth`

Custom setuptools hooks

`setup.py` can be extended easily:

```
...
entry_points = {
    "distutils.commands": [
        "build_jar = clamp.build:build_jar",
        "singlejar = clamp.build:singlejar",
    ], ...
```

Note: these commands are now available for any package that depends on clamp!

Anatomy of a command

Enabling support for `jython setup.py build_jar`, or any other custom command, requires the following attributes:

```
class build_jar(setuptools.Command):
  description = "create jar for clamped classes",
  user_options = [("output=",   "o", "write jar"),]

  def initialize_options(self): ...

  def finalize_options(self): ...

  def run(self): ...
```

Adding new scripts in Python's bin

`setup.py` uses the same `entry_points` to point to custom scripts:

```
...
entry_points = {
    "distutils.commands": [
        "build_jar = clamp.build:build_jar",
        "singlejar = clamp.build:singlejar",
    ],
    "console_scripts": [
        "singlejar = clamp.build:singlejar_command",
    ]
}
```

Console script example: singlejar

Script is just a standard *main* function, using arg parser of choice:

```
def singlejar_command():
  parser = argparse.ArgumentParser(...)
  parser.add_argument("--classpath", ...)
  parser.add_argument("--runpy", ...)
  args = parser.parse_args()
  if args.classpath:
    args.classpath = args.classpath.split(":")
  else:
    args.classpath = []
  clamp.build.create_singlejar(
    args.output, args.classpath, args.runpy)
```

Metaprogramming to be done

- Import hooks - intercept the import of Java annotations so they can be used as class and function decorators.
- Rewriting Java bytecode - with ASM to support annotations/type signatures as class decorators.
- More, much more!

Clamp resources

- Clamp project: https://github.com/jythontools/clamp
- Example project: https://github.com/jimbaker/clamped
- Documentation: http://clamp.readthedocs.org