Breaking the Language Barrier:

PJRmi and Python-Java Interoperability

April 2025

D E Shaw & Co

www.deshaw.com/library

We built PJRmi to allow for quick and seamless interoperation between Python and Java without the usual glue code that Python needs to leverage other languages. With PJRmi, Python code can call directly into Java, and vice versa, effectively turning Java into a Python extension. In addition to enabling new hybrid applications, this solution supports use cases including driving Java applications with Python scripts, exposing Java service interfaces to Python clients, command and control, and on-the-fly debugging. Given its general-purpose nature, we made PJRmi open source and freely available on GitHub.

The Challenge

When building software and systems over the past 35 years, we've learned the immense value of nimble programming. Research and infrastructure work benefit from flexible scripting languages, such as Python, which offer powerful data manipulation and rapid prototyping. By contrast, heavy computational tasks and larger-scale projects often favor compiled languages like C++ and Java, which excel at performance-critical operations and offer the structure and strictness essential to production systems.

Over time, many of our systems evolved into a mix of Python and Java (as well as other languages like C++, Rust, and Go), with certain components existing in one ecosystem or the other. This architecture, while functional, made for some trade-offs: certain Java elements could be more elegantly expressed in Python and vice versa.

Recognizing the limitations imposed by separate language ecosystems, we created PJRmi, a tool that melds Python and Java into one and enables developers to naturally write code spanning both languages.

```
>>> a = ArrayList(range(3))
>>> m = HashMap({i : i+1 for i in a})
>>> m.toString()
'{0=1, 1=2, 2=3}'
>>> for i in Arrays.asList([j for j in range(6)]):
...     if i not in m:
...         m[i] = i * 10
>>> str(m)
'{0=1, 1=2, 2=3, 3=30, 4=40, 5=50}'
```

Blurring the lines between Python and Java, where only the naming conventions make it obvious which objects are in which language.

A First Principles Approach

Many of PJRmi's features emerged from conversations with teams within our firm about their distinct needs and challenges. These discussions revealed a common desire for the topology to be invisible—for an application to span languages, hosts, and data centers without changing programming styles. This led us to develop solutions that work across our diverse technology stack while making our systems more adaptable.

```
>>> s = Integer.toString(123)
>>> s
'123'
>>> isinstance(s, str)
True
>>> rmi.is_instance_of(s, String)
True
>>> int(s)
123
>>> Integer.valueOf(s)
123
```

Primitives' class types from Java are boxed as their Python equivalents, joining the two type systems together.

Core Design Principles

1. Automatic Type Translation

- Handle Java-Python type conversions transparently
- Preserve semantic meaning across language boundaries
- Java types naturally duck-type as their Python equivalents

2. Natural Syntax

- Allow developers to use Pythonic patterns
- Expose Java's type safety and object model
- Minimize the cognitive overhead of crossing language boundaries

3. Performance Considerations

- Build in optimized support for common operations
- Minimize serialization overhead
- Enable efficient bulk operations
- Facilitate callback support for server processes

Building the Bridge

Bridging Python and Java goes beyond rudimentary communication; the core challenge lies in maintaining Java's strict typing and performance advantages while retaining Python's dynamic flexibility.

Developing a glue layer to blend these languages entailed creating an interface that felt native to developers in both environments. This required sophisticated handling of data types, method invocations, and error management to preserve the idiomatic behavior of each language. With that interface, our developers can now write applications using both languages rather than creating systems with separate Python and Java components.

Java

```
# From TECHNICAL.md - Method overloading example
public static String foo(Number n) { return "Number"; }
public static String foo(Integer n) { return "Integer"; }
```

Python

```
>>> Foo.foo(1)
'Integer'
>>> Foo.foo(1.0)
'Number'
```

PJRmi can handle Java's method overloading while maintaining Python's intuitive syntax and type flexibility.

PJRmi seamlessly integrates Python's dynamic qualities into Java's structured paradigm. Java objects become Python shim instances—lightweight wrapper classes that provide a natural Python interface to the Java layer. This allows Python developers to interact with Java counterparts as if they were native Python entities. We accomplish this using reflection to dynamically create mappings of Java object methods and members in Python object instances. For example, Java lists duck-type as Python lists, so developers can iterate over a Java list and access or modify elements, all while using familiar Python syntax. This purposeful blurring of boundaries facilitates interaction without sacrificing the advantages of either language's type system. Similarly, on the Java side, Python classes can duck-type as implementations of Java interfaces.

```
>>> Thread
pjrmi.java.lang.Thread
>>> class PythonRunnable(pjrmi.JavaProxyBase):
...     def run(self):
...         print("I ran!")
>>> runnable = PythonRunnable()
>>> thread = Thread(runnable)
>>> thread.start()
I ran!
```

PJRmi enables Python code to naturally implement Java interfaces, demonstrating the bridge between Python's duck-typing and Java's interface system.

From Specific to General

Because PJRmi creates a unified development environment instead of enforcing strict client-server boundaries, its impact extends beyond developer productivity gains. We've seen concrete transformations in what our teams can achieve:

- **Direct API Exposure to Client Users:** allowing Python clients to directly interact with Java services without intermediate RESTful services
- **Command and Control via CLI or Scripts:** enabling control over Java applications through Python scripts or the command line interface, enhancing flexibility and automation

In addition to supporting applications across both programming languages, PJRmi enables a single logical application to span different hosts, or even data centers, without changing programming style. This capability also benefits DevOps by allowing for dynamic execution—or even rewriting—of code in running Java processes.

PJRmi in Action

A typical workflow starts with prototyping a new feature in Python for rapid iteration and testing. Using PJRmi, this prototype directly interacts with existing Java components, accessing production data and services. As the feature matures and performance requirements become critical, teams can selectively reimplement performance-sensitive sections in Java. With PJRmi, this transition requires minimal changes to the overall system architecture.

Additionally, PJRmi allows developers to connect to running Java applications through Python's interactive shell. This enables real-time inspection and modification of Java objects. For instance, when faced with production issues, an engineer can examine internal states, adjust parameters, and test solutions without restarting the Java process. It's even possible to hot-patch code in a live Java process from a Python client. These capabilities significantly reduce downtime and accelerate problem resolution, especially in firefighting situations.

```python
# Call into Java from Python
map1.computeIfAbsent(
    100,
    # Java calls back into Python...
    lambda x:
        # ...and Python calls back-back into Java...
        map2.computeIfAbsent(
            x // 2,
            # ...which calls back-back into Python,
            # which calls back-back-back into Java.
            lambda y: Integer.valueOf(str(y)) + 1
        )
)
```

A rather contrived example shows how PJRmi enables Python functions to be used as Java lambdas, showcasing the integration of modern language features.

Open Sourcing PJRmi

Initially developed as an internal tool targeting a particular use case, PJRmi's generalized design has lent itself to broader use across our firm. In releasing it as open source software, our hope is to support the broader developer ecosystem, provide a window into our first principles thinking, and invite fresh perspectives that could enhance the tool further.

If you're interested in exploring PJRmi in greater depth, you'll find detailed examples, performance optimization approaches, and architectural insights on GitHub. The repository's examples and documentation expand on many of the concepts discussed here.

D E Shaw & Co

www.deshaw.com