Jython and Java Integration

Java integration is the heart of Jython application development. Most Jython developers are either Python developers that are looking to make use of the vast library of tools that the JVM has to offer, or Java developers that would like to utilize the Python language semantics without migrating to a completely different platform. The fact is that most Jython developers are using it so that they can take advantage of the vast libraries available to the Java world, and in order to do so there needs to be a certain amount of Java integration in the application. Whether you plan to use some of the existing Java libraries in your application, or you're interested in mixing some great Python code into your Java application, this chapter is geared to help with the integration.

This chapter will focus on integrating Java and Python, but it will explore several different angles on the topic. You will learn several techniques to make use Jython code within your Java applications. Perhaps you'd like to simplify your code a bit; this chapter will show you how to write certain portions of your code in Jython and others in Java so that you can make code as simple as possible.

You'll also learn how to make use of the many Java libraries within your Jython applications while using Pythonic syntax! Forget about coding those programs in Java: why not use Jython so that the Java implementations in the libraries are behind the scenes? This chapter will show how to write Python code and use the libraries directly from it.

Using Java Within Jython Applications

Making use of Java from within Jython applications is about as seamless as using external Jython modules within a Jython script. As you learned in Chapter 8, you can simply import the required Java classes and use them directly. Java classes can be called in the same fashion as Jython classes, and the same goes for method calling. You simply call a class method and pass parameters the same way you'd do in Python.

Type coercion occurs much as it does when using Jython in Java in order to seamlessly integrate the two languages. In the following table, you will see the Java types that are coerced into Python types and how they match up. Table 10-1 was taken from the Jython user guide.

Table 10-1. Python and Java Types

Java Type	**Python Type**
char	String(length of 1)
boolean	Integer(true = not zero)
byte, short, int, long	Integer
java.lang.String, byte[], char[]	String
java.lang.Class	JavaClass
Foo[]	Array(containing objects of class or subclass of Foo)

Java Type	**Python Type**
java.lang.Object	String
orb.python.core.PyObject	Unchanged
Foo	JavaInstance representing Java class Foo

Another thing to note about the utilization of Java within Jython is that there may be some naming conflicts. If a Java object conflicts with a Python object name, then you can simply fully qualify the Java object in order to ensure that the conflict is resolved. Another technique which was also discussed in Chapter 8 is making use of the 'as' keyword when importing in order to rename an imported piece of code.

In the next couple of examples, you will see some Java objects being imported and used from within Jython.

Listing 10-1. Using Java in Jython

```
>>> from java.lang import Math
>>> Math.max(4, 7)
7L
>>> Math.pow(10,5)
100000.0
>>> Math.round(8.75)
9L
>>> Math.abs(9.765)
9.765
>>> Math.abs(-9.765)
9.765
>>> from java.lang import System as javasystem
>>> javasystem.out.println("Hello")
Hello
```

Now let's create a Java object and use it from within a Jython application.

Beach.java

```
public class Beach {

    private String name;
    private String city;


    public Beach(String name, String city){
        this.name = name;
        this.city = city;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

}
```

Using Beach.java in Jython

```
>>> import Beach
>>> beach = Beach("Cocoa Beach","Cocoa Beach")
>>> beach.getName()
u'Cocoa Beach'
>>> print beach.getName()
Cocoa Beach
```

As we had learned in Chapter 8, one thing you'll need to do is ensure that the Java class you wish to use resides within your CLASSPATH. In the example above, I created a JAR file that contained the Beach class and then put that JAR on the CLASSPATH.

It is also possible to extend or subclass Java classes via Jython classes. This allows us to extend the functionality of a given Java class using Jython objects, which can be quite helpful at times. The next example shows a Jython class extending a Java class that includes some calculation functionality. The Jython class then adds another calculation method and makes use of the calculation methods from both the Java class and the Jython class.

Listing 10-2. Extending Java Classes

Calculator.java

```
/**
* Java calculator class that contains two simple methods
*/
public class Calculator {

    public Calculator(){

    }

    public double calculateTip(double cost, double tipPercentage){
        return cost * tipPercentage;
    }

    public double calculateTax(double cost, double taxPercentage){
        return cost * taxPercentage;
    }

}
```

JythonCalc.py

```python
import Calculator
from java.lang import Math

class JythonCalc(Calculator):
    def __init__(self):
        pass

    def calculateTotal(self, cost, tip, tax):
        return cost + self.calculateTip(tip) + self.calculateTax(tax)

if __name__ == "__main__":
    calc = JythonCalc()
    cost = 23.75
    tip = .15
    tax = .07
    print "Starting Cost: ", cost
    print "Tip Percentage: ", tip
    print "Tax Percentage: ", tax
    print Math.round(calc.calculateTotal(cost, tip, tax))
```

Result

```
Starting Cost: 23.75
Tip Percentage: 0.15
Tax Percentage: 0.07
29
```

Using Jython Within Java Applications

Often, it is handy to have the ability to make use of Jython from within a Java application. Perhaps there is a class that would be better implemented in Python syntax, such as a Javabean. Or maybe there is a handy piece of Jython code that would be useful within some Java logic. Whatever the case may be, there are several approaches you can use in order to achieve this combination of technologies. In this section, we'll cover some of the older techniques for using Jython within Java, and then go into the current and future best practices for doing this. In the end, you should have a good understanding for how to use a module, script, or even just a few lines of Jython within your Java application. You will also have an overall understanding for the way that Jython has evolved in this area.

Object Factories

Perhaps the most widely used technique used today for incorporating Jython code within Java applications is the object factory design pattern. This idea basically enables seamless integration between Java and Jython via the use of object factories. There are different implementations of the logic, but all of them do have the same result in the end.

Implementations of the object factory paradigm allow one to include Jython modules within Java applications without the use of an extra compilation step. Moreover, this technique allows for a clean integration of Jython and Java code through usage of Java interfaces. In this section, I will explain the main concept of the object factory technique and then I will show you various implementations.

Let’s take a look at an overview of the entire procedure from a high level. Say that you’d like to use one of your existing Jython modules as an object container within a Java application. Begin by coding a Java interface that contains definitions for those methods contained in the module that you’d like to expose to the Java application. Next, you would modify the Jython module to implement the newly coded Java interface. After this, code a Java factory class that would make the necessary conversions of the module from a PyObject into a Java object. Lastly, take the newly created Java object and use it as you wish. It may sound like a lot of steps in order to perform a simple task, but I think you’ll agree that it is not very difficult once you’ve seen it in action.

Over the next few sections, I will take you through different examples of the implementation. The first example is a simple and elegant approach that involves a one-to-one Jython object and factory mapping. In the second example, we’ll take a look at a very loosely coupled approach for working with object factories that basically allows one factory to be used for all Jython objects. Each of these methodologies has its own benefit and you can use the one that works best for you.

One-to-One Jython Object Factories

We will first discuss the notion of creating a separate object factory for each Jython object we wish to use. This one-to-one technique can prove to create lots of boilerplate code, but it has some advantages that we’ll take a closer look at later on. In order to utilize a Jython module using this technique, you must either ensure that the .py module is contained within your sys.path, or hard code the path to the module within your Java code. Let’s take a look at an example of this technique in use with a Java application that uses a Jython class representing a building.

Listing 10-3. Creating a One-To-One Object Factory

Building.py

```
# A python module that implements a Java interface to
# create a building object
from org.jython.book.interfaces import BuildingType
```

```
class Building(BuildingType):
    def __init__(self, name, address, id):
        self.name = name
        self.address = address
        self.id = id

    def getBuildingName(self):
        return self.name

    def getBuildingAddress(self):
        return self.address

    def getBuldingId(self):
        return self.id
```

We begin with a Jython module named *Building.py* that is placed somewhere on our sys.path. Now, we must first ensure that there are no name conflicts before doing so or we could see some quite unexpected results. It is usually a safe bet to place this file at the source root for your application unless you explicitly place the file in your sys.path elsewhere. You can see that our *Building.py* object is a simple container for holding building information. We must explicitly implement a Java interface within our Jython class. This will allow the PythonInterpreter to coerce our object later. Our second piece of code is the Java interface that we implemented in *Building.py*. As you can see from the code, the returning Jython types are going to be coerced into Java types, so we define our interface methods using the eventual Java types. Let's take a look at the Java interface next.

BuildingType.java

```
// Java interface for a building object
package org.jython.book.interfaces;

public interface BuildingType {

    public String getBuildingName();
    public String getBuildingAddress();
    public String getBuildingId();

}
```

Looking through the definitions contained within the Java interface, it is plain to see that the python module that subclasses it simply implements each of the definitions. If we wanted to change the python code a bit and add some code to one of the methods we could do so without touching the Java interface. The next piece of code that we need is a factory written in Java. This factory has the job of coercing the python module into a Java class.

BuildingFactory.java

```
/**
 *
 * Object Factory that is used to coerce python module into a
 * Java class
 */
package org.jython.book.util;

import org.jython.book.interfaces.BuildingType;
import org.python.core.PyObject;
import org.python.core.PyString;
import org.python.util.PythonInterpreter;

public class BuildingFactory {

    private PyObject buildingClass;

    /**
     * Create a new PythonInterpreter object, then use it to
     * execute some python code. In this case, we want to
     * import the python module that we will coerce.
     *
     * Once the module is imported than we obtain a reference to
     * it and assign the reference to a Java variable
     */

    public BuildingFactory() {
        PythonInterpreter interpreter = new PythonInterpreter();
        interpreter.exec("from Building import Building");
        buildingClass = interpreter.get("Building");
    }

    /**
     * The create method is responsible for performing the actual
     * coercion of the referenced python module into Java bytecode
     */

    public BuildingType create (String name, String location, String id) {

        PyObject buildingObject = buildingClass.__call__(new PyString(name),
                                                           new
PyString(location),
                                                           new PyString(id));
        return (BuildingType)buildingObject.__tojava__(BuildingType.class);
    }

}
```

The third piece of code in the example above plays a most important role, since this is the object factory that will coerce our Jython code into a resulting Java class. In the constructor, a new instance of the PythonInterpreter is created. We then utilize the interpreter to obtain a reference to our Jython object and store it into our PyObject. Next, there is a static method named *create* that will be called in order to coerce our Jython object into Java and return the resulting class. It does so by performing a *__call__* on the PyObject wrapper itself, and as you can see we have the ability to pass parameters to it if we like. The parameters must also be wrapped by PyObjects. The coercion takes place when the *__tojava__* method is called on the

PyObject wrapper. In order to make object implement our Java interface, we must pass the interface *EmployeeType.class* to the *__tojava__* call.

Main.java

```
package org.jython.book;

import org.jython.book.util.BuildingFactory;
import org.jython.book.interfaces.BuildingType;

public class Main {

    private static void print(BuildingType building) {
        System.out.println("Building Info: " +
                building.getBuildingId() + " " +
                building.getBuildingName() + " " +
                building.getBuildingAddress());
    }

    /**
     * Create three building objects by calling the create() method of
     * the factory.
     */

    public static void main(String[] args) {
        BuildingFactory factory = new BuildingFactory();
        print(factory.create("BUILDING-A", "100 WEST MAIN", "1"));
        print(factory.create("BUILDING-B", "110 WEST MAIN", "2"));
        print(factory.create("BUILDING-C", "120 WEST MAIN", "3"));

    }

}
```

The last bit of provided code, *Main.java*, shows how to make use of our factory. You can see that the factory takes care of all the heavy lifting and our implementation in *Main.java* is quite small. Simply call the *factory.create()* method to instantiate a new PyObject and coerce it into Java.

This procedure for using the object factory design has the benefit of maintaining complete awareness of the Jython object from within Java code. In other words, creating a separate factory for each Jython object allows for the use of passing arguments into the constructor of the Jython object. Since the factory is being designed for a specific Jython object, we can code the *__call__* on the PyObject with specific arguments that will be passed into the new constructor of the coerced Jython object. Not only does this allow for passing arguments into the constructor, but also increases the potential for good documentation of the object since the Java developer will know exactly what the new constructor will look like. The procedures performed in this subsection are probably the most frequently used throughout the Jython community. In the next section, we'll take a look at the same technique applied to a generic object factory that can be used by any Jython object.

Summary of One-to-One Object Factory

The key to this design pattern is the creation of a factory method that utilizes PythonInterpreter in order to load the desired Jython module. Once the factory has loaded the module via PythonInterpreter, it creates a PyObject instance of the module. Lastly, the factory coerces the PyObject into Java code using the PyObject __tojava__ method.

Overall, the idea is not very difficult to implement and relatively straightforward. However, the different implementations come into play when it comes to passing references for the Jython module and a corresponding Java interface. It is important to note that the factory takes care of instantiating the Jython object and translating it into Java. All work that is performed against the resulting Java object is coded against a corresponding Java interface. This is a great design because it allows us to change the Jython code implementation if we wish without altering the definition contained within the interface. The Java code can be compiled once and we can change the Jython code at will without breaking the application.

Making Use of a Loosely Coupled Object Factory

The object factory design does not have to be implemented using a one to one strategy such as that depicted in the example above. It is possible to design the factory in such a way that it is generic enough to be utilized for any Jython object. This technique allows for less boilerplate coding as you only require one Singleton factory that can be used for all Jython objects. It also allows for ease of use as you can separate the object factory logic into its own project and then apply it wherever you'd like. For instance, I've created a project named PlyJy (http://kenai.com/projects/plyjy) that basically contains a Jython object factory that can be used in any Java application in order to create Jython objects from Java without worrying about the factory. You can go to Kenai and download it now to begin learning about loosely coupled object factories. In this section we'll take a look at the design behind this project and how it works.

Let's take a look at the same example from above and apply the loosely coupled object factory design. You will notice that this technique forces the Java developer to do a bit more work when creating the object from the factory, but it has the advantage of saving the time that is spent to create a separate factory for each Jython object. You can also see that now we need to code setters into our Jython object and expose them via the Java interface as we can no longer make use of the constructor for passing arguments into the object since the loosely coupled factory makes a generic __*call*__ on the PyObject.

Listing 10-4. Using a Loosely Coupled Object Factory

Building.py

```
from org.jython.book.interfaces import BuildingType
# Building object that subclasses a Java interface

class Building(BuildingType):
    def __init__(self):
        self.name = None
        self.address = None
        self.id = -1

    def getBuildingName(self):
        return self.name

    def setBuildingName(self, name):
        self.name = name;

    def getBuildingAddress(self):
        return self.address

    def setBuildingAddress(self, address)
        self.address = address

    def getBuildingId(self):
        return self.id

    def setBuildingId(self, id):
        self.id = id
```

If we follow this paradigm then you can see that our Jython module must be coded a bit differently than it was in our one-to-one example. The main differences are in the initializer as it no longer takes any arguments, and we therefore have coded setter methods into our object. The rest of the concept still holds true in that we must implement a Java interface that will expose those methods we wish to invoke from within our Java application. In this case, we coded the*BuildingType.java* interface and included the necessary setter definitions so that we have a way to load our class with values.

BuildingType.java

```
package org.jython.book.interfaces;

/**
 * Java interface defining getters and setters
 */

public interface BuildingType {

 public String getBuildingName();
 public String getBuildingAddress();
 public int getBuildingId();
 public void setBuildingName(String name);
 public void setBuildingAddress(String address);
 public void setBuildingId(int id);

}
```

Our next step is to code a loosely coupled object. If you take a look at the code in the *JythonObjectFactory.java* class you will see that it is a singleton; that is it can only be instantiated one time. The important method to look at is *createObject()* as it does all of the work.

JythonObjectFactory.java

```
import java.util.logging.Level;
import java.util.logging.Logger;
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;

/**
 * Object factory implementation that is defined
 * in a generic fashion.
 *
 */

public class JythonObjectFactory {
    private static JythonObjectFactory instance = null;
    private static PyObject pyObject = null;

    protected JythonObjectFactory() {

    }
    /**
     * Create a singleton object. Only allow one instance to be created
     */
    public static JythonObjectFactory getInstance(){
        if(instance == null){
            instance = new JythonObjectFactory();
        }

        return instance;
    }

    /**
     * The createObject() method is responsible for the actual creation of
the
     * Jython object into Java bytecode.
     */
    public static Object createObject(Object interfaceType, String
moduleName){
        Object javaInt = null;
        // Create a PythonInterpreter object and import our Jython module
        // to obtain a reference.
        PythonInterpreter interpreter = new PythonInterpreter();
        interpreter.exec("from " + moduleName + " import " + moduleName);

        pyObject = interpreter.get(moduleName);

        try {
            // Create a new object reference of the Jython module and
            // store into PyObject.
            PyObject newObj = pyObject.__call__();
            // Call __tojava__ method on the new object along with the
interface name
```

```
            // to create the java bytecode
            javaInt =
newObj.__tojava__(Class.forName(interfaceType.toString().substring(
                        interfaceType.toString().indexOf(" ")+1,
                        interfaceType.toString().length())));
        } catch (ClassNotFoundException ex) {

Logger.getLogger(JythonObjectFactory.class.getName()).log(Level.SEVERE, null,
ex);
        }

        return javaInt;
    }

}
```

As you can see from the code, the PythonInterpreter is responsible for obtaining a reference to the Jython object name that we pass as a String value into the method. Once the PythonInterpreter has obtained the object and stored it into a PyObject, its *__call__()* method is invoked without any parameters. This will retrieve an empty object that is then stored into another PyObject referenced by *newObj*. Lastly, our newly obtained object is coerced into Java code by calling the *__tojava__()* method which takes the fully qualified name of the Java interface we've implemented with our Jython object. The new Java object is then returned.

Main.java

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jythonbook.interfaces.BuildingType;
import org.jybhonbook.factory.JythonObjectFactory;

public class Main {

    public static void main(String[] args) {

    // Obtain an instance of the object factory
    JythonObjectFactory factory = JythonObjectFactory.getInstance();

    // Call the createObject() method on the object factory by
    // passing the Java interface and the name of the Jython module
    // in String format. The returning object is casted to the the same
    // type as the Java interface and stored into a variable.
    BuildingType building = (BuildingType) factory.createObject(
        BuildingType.class, "Building");
    // Populate the object with values using the setter methods
    building.setBuildingName("BUIDING-A");
    building.setBuildingAddress("100 MAIN ST.");
    building.setBuildingId(1);
    System.out.println(building.getBuildingId() + " " +
building.getBuildingName() + " " +
        building.getBuildingAddress());
    }

}
```

Taking a look at the *Main.java* code, you can see that the factory is instantiated or referenced via the use of the *JythonObjectFactory.getInstance()*. Once we have an instance of the factory, the *createObject(**Interface, String)* is called passing the interface and a string representation of the module name we wish to use. The code must cast the coerced object using the interface as well. This example assumes that the object resides somewhere on your sys.path, otherwise you can use the *createObjectFromPath(**Interface, String)* that accepts the string representation for the path to the module we'd like to coerce. This is of course not a preferred technique since it will now include hard-coded paths, but it can be useful to apply this technique for testing purposes. For example if you've got two Jython modules coded and one of them contains a different object implementation for testing purposes, then this technique will allow you to point to the test module.

More Efficient Version of Loosely Coupled Object Factory

Another similar, yet, more refined implementation omits the use of PythonInterpreter and instead makes use of PySystemState. Why would we want another implementation that produces the same results? Well, there are a couple of reasons. The loosely coupled object factory design I described in the beginning of this section instantiates the PythonInterpreter and then makes calls against it. This can cause a decrease in performance, as it is quite expensive to use the interpreter. On the other hand, we can make use of PySystemState and save ourselves the trouble of incurring extra overhead making calls to the interpreter. Not only does the next example show how to utilize this technique, but it also shows how we can make calls upon the coerced object and pass arguments at the same time.

Listing 10-5. Use PySystemState to Code a Loosely Coupled Factory

JythonObjectFactory.java

```
package org.jython.book.util;

import org.python.core.Py;
import org.python.core.PyObject;
import org.python.core.PySystemState;

/**
 * Jython Object Factory using PySystemState
 */
public class JythonObjectFactory {

 private final Class interfaceType;
 private final PyObject klass;

 // Constructor obtains a reference to the importer, module, and the class
name
```

```
 public JythonObjectFactory(PySystemState state, Class interfaceType, String
moduleName, String className) {
     this.interfaceType = interfaceType;
     PyObject importer =
state.getBuiltins().__getitem__(Py.newString("__import__"));
     PyObject module = importer.__call__(Py.newString(moduleName));
     klass = module.__getattr__(className);
     System.err.println("module=" + module + ",class=" + klass);
 }

 // This constructor passes through to the other constructor
 public JythonObjectFactory(Class interfaceType, String moduleName, String
className) {
     this(new PySystemState(), interfaceType, moduleName, className);
 }

 // All of the followng methods return
 // a coerced Jython object based upon the pieces of information
 // that were passed into the factory. The differences are
 // between them are the number of arguments that can be passed
 // in as arguents to the object.

 public Object createObject() {
     return klass.__call__().__tojava__(interfaceType);
 }


 public Object createObject(Object arg1) {
     return klass.__call__(Py.java2py(arg1)).__tojava__(interfaceType);
 }

 public Object createObject(Object arg1, Object arg2) {
     return klass.__call__(Py.java2py(arg1),
Py.java2py(arg2)).__tojava__(interfaceType);
 }

 public Object createObject(Object arg1, Object arg2, Object arg3)
 {
     return klass.__call__(Py.java2py(arg1), Py.java2py(arg2),
         Py.java2py(arg3)).__tojava__(interfaceType);
 }

 public Object createObject(Object args[], String keywords[]) {
     PyObject convertedArgs[] = new PyObject[args.length];
     for (int i = 0; i < args.length; i++) {
         convertedArgs[i] = Py.java2py(args[i]);
     }

     return klass.__call__(convertedArgs,
keywords).__tojava__(interfaceType);
 }

 public Object createObject(Object... args) {
     return createObject(args, Py.NoKeywords);
 }
```

```
}
```

Main.java

```
import org.jython.book.interfaces.BuildingType;
import org.jython.book.util.JythonObjectFactory;

public class Main{

    public static void main(String args[]) {

        JythonObjectFactory factory = new JythonObjectFactory(
            BuildingType.class, "building", "Building");

        BuildingType building = (BuildingType) factory.createObject();

        building.setBuildingName("BUIDING-A");
        building.setBuildingAddress("100 MAIN ST.");
        building.setBuildingId(1);

        System.out.println(building.getBuildingId() + " " +
            building.getBuildingName() + " " +
            building.getBuildingAddress());
    }

}
```

As you can see from the code, there are quite a few differences from the object factory implementation shown previously. First, you can see that the instantiation of the object factory requires different arguments. In this case, we pass in the interface, module, and class name. Next, you can see that the PySystemState obtains a reference to the importer PyObject. The importer then makes a __call__ to the module we've requested. The requested module must be contained somewhere on the sys.path. Lastly, we obtain a reference to our class by calling the __getattr__ method on the module. We can now use the returned class to perform the coercion of our Jython object into Java. As mentioned previously, you'll note that this particular implementation includes several createObject() variations allowing one to pass arguments to the module when it is being called. This, in effect, gives us the ability to pass arguments into the initializer of the Jython object.

Which object factory is best? Your choice, depending upon the situation you're application is encountering. Bottom line is that there are several ways to perform the object factory design and they all allow seamless use of Jython objects from within Java code.

Now that we have a coerced Jython object, we can go ahead and utilize the methods that have been defined in the Java interface. As you can see, the simple example above sets a few values and then prints out the object values. Hopefully you can see how easy it is to create a single object factory that we can be use for any Jython object rather than just one.

Returning __doc__ Strings

It is also very easy to obtain the *__doc__* string from any of your Jython classes by coding an accessor method on the object itself. We'll add some code to the building object that was used in the previous examples. It doesn't matter what type of factory you decide to work with, this trick will work with both.

Listing 10-6. __doc__ Strings

Building.py

```python
from org.jython.book.interfaces import BuildingType
# Notice the doc string that has been added after the class definition below

class Building(BuildingType):
    ''' Class to hold building objects '''

    def __init__(self):
        self.name = None
        self.address = None
        self.id = -1

    def getBuildingName(self):
        return self.name

    def setBuildingName(self, name):
        self.name = name;

    def getBuildingAddress(self):
        return self.address

    def setBuildingAddress(self, address):
        self.address = address

    def getBuildingId(self):
        return self.id

    def setBuildingId(self, id):
        self.id = id

    def getDoc(self):
        return self.__doc__
```

BuildingType.java

```java
package org.jython.book.interfaces;

public interface BuildingType {

    public String getBuildingName();
    public String getBuildingAddress();
    public int getBuildingId();
    public void setBuildingName(String name);
    public void setBuildingAddress(String address);
    public void setBuildingId(int id);
    public String getDoc();

}
```

Main.java

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jython.book.interfaces.BuildingType;
import org.plyjy.factory.JythonObjectFactory;

public class Main {

    public static void main(String[] args) {

        JythonObjectFactory factory = JythonObjectFactory.getInstance();
        BuildingType building = (BuildingType) factory.createObject(
            BuildingType.class, "Building");
        building.setBuildingName("BUIDING-A");
        building.setBuildingAddress("100 MAIN ST.");
        building.setBuildingId(1);
        System.out.println(building.getBuildingId() + " " +
            building.getBuildingName() + " " +
        building.getBuildingAddress());

        // It is easy to print out the documentation for our Jython object
        System.out.println(building.getDoc());

    }
}
```

Result:

```
1 BUIDING-A 100 MAIN ST.
Class to hold building objects
```

Applying the Design to Different Object Types

This design will work with all object types, not just plain old Jython objects. In the following example, the Jython module is a class containing a simple calculator method. The factory coercion works the same way, and the result is a Jython class that is converted into Java.

Listing 10-7. Different Method Types

CostCalculator.py

```
from org.jython.book.interfaces import CostCalculatorType

class CostCalculator(CostCalculatorType, object):
    ''' Cost Calculator Utility '''

    def __init__(self):
        print 'Initializing'
        pass

    # The implementation for the definition contained in the Java interface
    def calculateCost(self, salePrice, tax):
        return salePrice + (salePrice * tax)
```

CostCalculatorType.java

```
package org.jython.book.interfaces;

public interface CostCalculatorType {

    public double calculateCost(double salePrice, double tax);

}
```

Main.java

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.jython.book.interfaces.CostCalculatorType;
import org.plyjy.factory.JythonObjectFactory;

public class Main {

    public static void main(String[] args) {

        // Create factory and coerce Jython calculator object
        JythonObjectFactory factory = JythonObjectFactory.getInstance();
        CostCalculatorType costCalc = (CostCalculatorType)
            factory.createObject(CostCalculatorType.class, "CostCalculator");
        System.out.println(costCalc.calculateCost(25.96, .07));

    }
}
```

Result

```
Initializing
27.7772
```

A Bit of History Prior to Jython 2.5, the standard distribution of Jython included a utility known as jythonc. Its main purpose was to provide the ability to convert Python modules into Java classes so that Java applications could seamlessly make use of Python code, albeit in a roundabout fashion. jythonc actually compiles the Jython code down into Java

.class files and then the classes are utilized within the Java application. This utility could also be used to freeze code modules, create jar files, and to perform other tasks depending upon which options were used. This technique is no longer the recommended approach for utilizing Jython within Java applications. As a matter of fact, jythonc is no longer packaged with the Jython distribution beginning with the 2.5 release.

In order for jythonc to take a Jython class and turn it into a corresponding Java class, it had to adhere to a few standards. First, the Jython class had to subclass a Java object, either a class or interface. It also had to do one of the following: override a Java method, implement a Java method, or create a new method using a signature.

While this method worked well and did what it was meant to do, it caused a separation between the Jython code and the Java code. The step of using jythonc to compile Jython into Java is clean, yet, it creates a rift in the development process. Code should work seamlessly without the need for separate compilation procedure. One should have the ability to utilize Jython classes and modules from within a Java application by reference only, and without a special compiler in between. There have been some significant advances in this area, and many of the newer techniques have been discussed in this chapter.

JSR-223

With the release of Java SE 6 came a new advantage for dynamic languages on the JVM. JSR-223 enables dynamic languages to be callable via Java in a seamless manner. Although this method of accessing Jython code is not quite as flexible as using an object factory, it is quite useful for running short Jython scripts from within Java code. The scripting project (https://scripting.dev.java.net/) contains many engines that can be used to run different languages within Java. In order to run the Jython engine, you must obtain jython-engine.jar from the scripting project and place it into your classpath. You must also place jython.jar in the classpath, and it does not yet function with Jython 2.5 so Jython 2.5.1 must be used.

Below is a small example showing the utilization of the scripting engine.

Listing 10-8. Using JSR-223

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws ScriptException {
        ScriptEngine engine = new
ScriptEngineManager().getEngineByName("python");

        // Using the eval() method on the engine causes a direct
        // interpretataion and execution of the code string passed into it
        engine.eval("import sys");
        engine.eval("print sys");

        // Using the put() method allows one to place values into
        // specified variables within the engine
        engine.put("a", "42");

        // As you can see, once the variable has been set with
        // a value by using the put() method, we an issue eval statements
        // to use it.
        engine.eval("print a");
        engine.eval("x = 2 + 2");

        // Using the get() method allows one to obtain the value
        // of a specified variable from the engine instance
        Object x = engine.get("x");
        System.out.println("x: " + x);
    }

}
```

Next, we see the result of running the application. The first two lines are automatically generated when the Jython interpreter is initiated; they display the JAR filescontained within the CLASSPATH. Following those lines, we see the actual program output.

Result

```
*sys-package-mgr*: processing new jar,'/jsr223-engines/jython/build/jython-
engine.jar'
*sys-package-mgr*: processing modified
jar,'/System/Library/Java/Extensions/QTJava.zip'
sys module
42
x: 4
```

Utilizing PythonInterpreter

A similar technique to JSR-223 for embedding Jython is making use of the PythonInterpreter directly. This style of embedding code is very similar to making use of a scripting engine, but it has the advantage of working with Jython 2.5. Another advantage is that the PythonInterpreter enables you to make use of PyObjects directly. In order to make use of the PythonInterpreter technique, you only need to have jython.jar in your classpath; there is no need to have an extra engine involved.

Listing 10-9. Using PythonInterpreter

```
import org.python.core.PyException;
import org.python.core.PyInteger;
import org.python.core.PyObject;
import org.python.util.PythonInterpreter;

public class Main {

    /**
    * @param args the command line arguments
    */
    public static void main(String[] args) throws PyException {

        // Create an instance of the PythonInterpreter
        PythonInterpreter interp = new PythonInterpreter();

        // The exec() method executes strings of code
        interp.exec("import sys");
        interp.exec("print sys");

        // Set variable values within the PythonInterpreter instance
        interp.set("a", new PyInteger(42));
        interp.exec("print a");
        interp.exec("x = 2+2");

        // Obtain the value of an object from the PythonInterpreter and store
it
        // into a PyObject.
        PyObject x = interp.get("x");
        System.out.println("x: " + x);
    }

}
```

In the class above, we make use of the PythonInterpreter to execute Python code within the Java class. First, we create an instance of the PythonInterpreter object. Next, we make exec() calls against it to execute strings of code passed into it. Next we use the set() method in order to set variables within the interpreter instance. Lastly, we obtain a copy of the object that is stored in the variable x within the interpreter. We must store that object as a PyObject in our Java code.

Results

```
<module 'sys' (built-in)>
42
x: 4
```

The following is a list of methods available for use within a PythonInterpreter object along with a description of functionality.

Table 10-2. PythonInterpreter Methods**

Method	Description
setIn(PyObject)	Set the Python object to use for the standard input stream

Method	**Description**
setIn(java.io.Reader)	Set a java.io.Reader to use for the standard input stream
setIn(java.io.InputStream)	Set a java.io.InputStream to use for the standard input stream
setOut(PyObject)	Set the Python object to use for the standard output stream
setOut(java.io.Writer)	Set the java.io.Writer to use for the standard output stream
setOut(java,io.OutputStream)	Set the java.io.OutputStream to use for the standard output stream
setErr(PyObject)	Set a Python error object to use for the standard error stream
setErr(java.io.Writer	Set a java.io.Writer to use for the standard error stream
setErr(java.io.OutputStream)	Set a java.io.OutputStream to use for the standard error stream
eval(String)	Evaluate a string as Python source and return the result
eval(PyObject)	Evaluate a Python code object and return the result
exec(String)	Execute a Python source string in the local namespace
exec(PyObject)	Execute a Python code object in the local namespace
execfile(String filename)	Execute a file of Python source in the local namespace
execfile(java.io.InputStream)	Execute an input stream of Python source in the local namespace
compile(String)	Compile a Python source string as an expression or module
compile(script, filename)	Compile a script of Python source as an expression or module
set(String name, Object value)	Set a variable of Object type in the local namespace
set(String name, PyObject value)	Set a variable of PyObject type in the local namespace
get(String)	Get the value of a variable in the local namespace
get(String name, Class<T> javaclass	Get the value of a variable in the local namespace. The value will be returned as an instance of the given Java class.

Summary

Integrating Jython and Java is really at the heart of the Jython language. Using Java within Jython works just as we as adding other Jython modules; both integrate seamlessly. What makes this nice is that now we can use the full set of libraries and APIs available to Java from our Jython applications. Having the ability of using Java within Jython also provides the advantage of writing Java code in the Python syntax.

Utilizing design patterns such as the Jython object factory, we can also harness our Jython code from within Java applications. Although *jythonc* is no longer part of the Jython distribution, we can still effectively use Jython from within Java. There are object factory examples available, as well as projects such as PlyJy (http://kenai.com/projects/plyjy) that give the ability to use object factories by simply including a JAR in your Java application.

We learned that there are more ways to use Jython from within Java as well. The Java language added scripting language support with JSR-223 with the release of Java 6. Using a jython engine, we can make use of the JSR-223 dialect to sprinkle Jython code into our Java applications. Similarly, the PythonInterpreter can be used from within Java code to invoke Jython. Also keep an eye on projects such as Clamp (http://github.com/groves/clamp/tree/master): the Clamp project has the goal to make use of annotations in order to create Java classes from Jython classes. It will be exciting to see where this project goes, and it will be documented once the project has been completed.

Source: http://www.jython.org/jythonbook/en/1.0/JythonAndJavaIntegration.html