Java interop with ABCL, a practical example

Alessio Stalla
ManyDesigns s.r.l.
alessiostalla@gmail.com

ABSTRACT
In this paper, we give an overview of the Java interoperability features in the Armed Bear Common Lisp implementation, and we present an example of the application of some of those features to a real-world software project.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – Classes and objects, Inheritance, Frameworks.

General Terms
Languages, Experimentation.

Keywords
JVM, ABCL, Lisp, Java, interoperability.

1. INTRODUCTION
The Java Virtual Machine (JVM) [1] is a software platform that is widely used on servers, mobile devices, and, to a minor extent, desktops. Originally designed to host the now popular Java language [2], a mostly-static single-inheritance OO language in the C-like family, thanks to its dynamic features (which Java only partially benefits from), highly-optimized Just-In-Time compiler and garbage collectors, huge standard library and vast assortment of third-party libraries, frameworks and components, both commercial and open-source, the JVM nowadays hosts many static and dynamic languages, such as Scala[3], Groovy[4], JRuby[5], including several in the Lisp family: at least two Common Lisp implementations (ABCL[6] and CLForJava[7]), several Scheme implementations including the popular Kawa [8], and Clojure [9], a new Lisp dialect which has grown to respectable popularity.

We will focus in particular on the Armed Bear Common Lisp implementation (ABCL, [6]). We will give an overview of the facilities provided by ABCL to integrate Lisp and Java in a single software project. We will then show a proof-of-concept integration with a real-world application. We will also draw comparisons with Groovy, a popular scripting language for the Java platform.

2. ABCL
Armed Bear Common Lisp is an implementation of the Common Lisp standard targeting the Java Virtual Machine. ABCL is written in a combination of Java and Lisp and features both an interpreter and a compiler to JVM bytecode, limited debugging capabilities, SLIME integration, and facilities for Java interop which will be detailed in the following sections.

ABCL was originally developed by Peter Graves starting from 2002, as part of the J editor [10]. In 2008, Erik Huelsmann became the new maintainer of the project, which was subsequently moved from SourceForge to common-lisp.net and separated from the J editor. Over the years Huelsmann, with the help of other people, brought ABCL to be a stable, mature Common Lisp implementation with very few deviations from the standard, with recent release 1.0.1 failing only ~20 tests out of ~21700 in the ANSI compliance test suite by P. F. Dietz [11]. ABCL 1.0.0 was announced at the European Common Lisp Meeting 2011 in Amsterdam.

3. JAVA INTEROP OVERVIEW
3.1 Calling Java from Lisp
The JVM provides facilities for introspecting classes and objects, dynamically accessing fields (read/write) and calling methods in arbitrary objects. Collectively these facilities are known as the Reflection API [12].

ABCL's Java FFI (Foreign Function Interface) is built upon the reflection API. Conceptually this makes it independent from any compiler support, although some optimizations are only possible by specially handling certain forms in the compiler (ABCL implements only some of the possible optimizations, which are out of the scope of this paper). Additionally, such reliance on the reflection API makes the Java FFI quite different from other Lisp FFIs (such as the popular CFFI [13], which allows to interface Lisp with C/C++ on a variety of platforms): whereas generally FFIs require the user to declare in advance the foreign data types and function signatures accessed by Lisp code, ABCL offers the option to dynamically select methods and fields giving the minimum amount of information, at the cost of a higher runtime cost per foreign call or field access. When performance is important, it is still possible to provide all the necessary type information to eliminate dynamic dispatch, although, as previously observed, there is still room for improvement in the compiler to further optimize certain calls.

We illustrate with a small example a typical use of the Java FFI (taken from the Dynaspring library), without going into details:

```lisp
(defun default-resource-loader ()
  (if *bean-definition-reader*
    (jcall "getResourceLoader" *bean-definition-reader*)
    (jnew "org...DefaultResourceLoader")))
```
ABCL's companion contrib project provides a library called JSS, originally developed by Alan Ruttenberg, which further simplifies the syntax for calling into Java. JSS is out of the scope of this paper, but we encourage every interested party to delve into the topic.

3.2 Calling Lisp from Java
ABCL is itself partly implemented in Java, to the extent that most fundamental Lisp types are defined by a corresponding Java class. It is thus possible to directly manipulate Lisp objects in Java, mimicking the style one would use in Lisp. Naturally, Java not being Lisp, such style is definitely more verbose than its Lisp counterpart. Additionally, direct manipulation of certain complex objects – for example structures and standard-objects – requires knowledge of their internal representation, and is therefore hard, low level, and too much dependent on implementation details.

Fortunately, ABCL provides a high-level interface to invoke Lisp from Java and perform common tasks. This interface follows a specification known as the Java Scripting Interface or JSR-223 [14] which makes ABCL a first-class citizen among scripting languages on the Java platform. The JSR-223 API is such that, for simple enough tasks (loading or compiling a file, calling a function, accessing a variable, etc.), no compile-time dependency on ABCL is required in a Java program. This API also automatically performs useful conversions (from Java to Lisp and back) whenever appropriate, for example when invoking a function.

Anyway, for certain Lisp constructs and idioms that do not exist in Java (such as binding special variables or condition handlers), some manual work is left to the programmer. For example, the following Java code executes a function with *print-circle* bound to T:

```java
LispThread t = LispThread.currentThread();
SpecialBindingsMark m = t.markSpecialBindings();
t.bindSpecial(Symbol.PRINT_CIRCLE, Symbol.T);
try {
    return function.execute(args);
} finally {
    t.resetSpecialBindings(mark);
}
```

In our experience, in real-world projects it is preferable to use a mixed style of high-level JSR-223 calls and lower-level direct access to Lisp objects, depending on the task.

3.3 Beyond FFI: extending Java in Lisp
A feature which is, to our knowledge, unique of the ABCL FFI (as compared to other FFIs available on Common Lisp implementations), is the ability to integrate with the host platform to the point of being able to extend Java classes with Lisp code. This feature actually comes in two flavours, detailed below.

3.3.1 Implementing interfaces
With ABCL, it is possible to implement one or more Java interfaces using Lisp functions. It is possible, in the JVM, an interface is a special kind of abstract class that defines a contract in the form of a set of methods that subclasses must implement. Contrarily to regular classes, multiple inheritance of interfaces is possible. Interfaces are commonly used to abstract all kinds of behavior, and especially when dealing with event listeners, visitor objects, callbacks and the like. The advent of some advanced techniques (e.g., certain popular forms of Aspect-Oriented Programming as provided by the Spring Framework [15]), as well as the introduction of the so-called Project Lambda [16] in the forthcoming Java 8 specification (aiming to provide syntax sugar for anonymous function-like constructs), have made and will continue to make interfaces a key feature of the Java language and a principal point of integration with many libraries and frameworks. ABCL uses a native feature of the Java Reflection API, called dynamic proxies [17], to allow the implementation of interfaces in several ways, all revolving around the use of Lisp functions as the implementations of Java methods – for example, one built-in facility permits to give an implementation of a JVM interface with a Lisp hash-table whose keys are method names and whose values are the functions that constitute the implementation. User code can provide custom extensions of the interface implementation facility by specializing a generic function.

3.3.2 Extending classes
Although programming to interfaces is considered a best practice in Java, it's not always possible to achieve integration using interfaces alone. Some libraries or frameworks mandate the user to extend certain third-party classes, for example to inherit important built-in functionality, or to customize and extend those libraries themselves, or simply due to poor design. Also, with the introduction of annotations in Java 5 [18], a system for attaching metadata to various program elements, it has become common practice to use classes as holders of various types of configuration information, for example in web frameworks such as Stripes[19] or other types of libraries (ORMs like Hibernate[20], application frameworks like Spring[21], etc.). ABCL tackles the aforementioned problem by giving the user the ability to extend Java classes in Lisp, principally by providing the implementation of methods with Lisp functions, but also, if required, by adding fields and placing annotation metadata where it is possible to do so. This feature, known as runtime-class, has been only recently re-added to ABCL (it had been lost for quite a long time), and at the time of writing it is still new and experimental.

The runtime-class feature will be demonstrated, along with other aspects of the FFI, in the example which is the subject of the next section.

4. A PRACTICAL EXAMPLE
We will examine a very simple example of the use of ABCL as the extension language of an existing real-world Java web application. The application was chosen taking into consideration the author's knowledge of it and the fact that it already has provisions for being customized via scripts written in Groovy, a popular dynamic language for the JVM which has easy Java integration, down to source compatibility, as a core feature. We believe that a comparison with Groovy will show Lisp's strong points as an extension language.

4.1 ManyDesigns Portofino
The target of our example is ManyDesigns Portofino 4, the flagship product of ManyDesigns s.r.l. It is a model-driven application framework tailored at building database-driven enterprise web applications running on the JVM, with extensibility as one of the core goals. At the time of writing, ManyDesigns Portofino 4 is being developed internally and sold commercially to a few selected clients, but it will eventually be released as open source.
For the purpose of this paper, it suffices to say that Portofino 4 applications are organized as a tree of pages of various types. Being based on the Stripes MVC\(^1\) web framework, each page is implemented with an action class (that defines the events it can respond to and its consequent behaviour) plus one or more views in the form of JSP\(^3\) files.

The tree of gives form to the URLs exposed by the application and to its navigation panel, and corresponds to a tree of directories physically stored on the file system. Each directory contains a couple of configuration files and, most importantly, an action.groovy file defining a class. That class defines the type of the page (SCRUD\(^3\) over a database table, Chart page, Text page, etc.) and is also the main point of extension to customize it. For example, a standard CRUD page will have at an action.groovy file like:

```groovy
[imports elided]
class foo extends CrudAction {}
```

Each action can respond to HTTP requests via handler methods with the signature

```java
public Resolution method()
```

if the request contains a GET or POST parameter named X and a method with that signature and named X exists in the action, it will be called; the resulting Resolution object will be used by the Stripes web framework, on which Portofino 4 is built, to determine the next view to show to the user.

Via annotations, access to action methods can be restricted to only users with certain privileges, for example:

```
@RequiresPermissions(level = AccessLevel.VIEW)
public Resolution method() { ... }
```

Annotations can also be used to expose a given method as a user-visible button on the page. The same method can back up more than one button in different areas of the page. For example:

```
@Button(list = "crud-edit", key = "my.button")
public Resolution method() { ... }
```

Additional annotations can be used to place guards on operations; the corresponding buttons are disabled or hidden, and in any case the guarded method is not allowed to be invoked, if the guards preconditions are not met. This feature will not be interested by our little example.

### 4.2 Integrating Lisp in Portofino

Standing the premises outlined in the previous section, it appears clearly that a Lisp integrated in Portofino 4 needs to be able to extend Java classes so we can use them as custom actions, and to annotate them so we can control access rights and buttons.

Even though the file-based nature of the framework suits Groovy better than Lisp, where a somewhat more image-based style is generally preferred, we won't touch this aspect. We speculate that migrating to a more Lisp-friendly style of development and interaction would not be particularly complex, but to keep the example simple we have decided to supersede about it.

We will omit for brevity the incantations needed to add ABCL as a dependency to Portofino using Maven. Once ABCL is added as a library, we can start patching the application to replace Groovy with Lisp (a possible improvement over this basic example would be having Groovy, Lisp, and possibly other languages coexist in the same application). We will add a method to load a Lisp file defining a custom action class to the already existing ScriptingUtil class, which has a few utility methods for loading Groovy code:

```java
public static final Abc1ScriptEngine ABCL = (Abc1ScriptEngine) new Abc1ScriptEngineFactory().getScriptEngine();
```

```java
public static Class<?> getLispClass(File storageDirFile, String id) throws IOException {  
  File scriptFile = getLispScriptFile(storageDirFile, id);  
  if(!scriptFile.exists()) {  
    return null;  
  }  
  FileReader fr = new FileReader(scriptFile);  
  try {  
    Object result = ABCL.eval(fr);  
    return (Class) result;  
  } catch(ScriptException e) {  
    throw new RuntimeException(e);  
  } finally {  
    IOUtils.closeQuietly(fr);  
  }
```

Then, replacing the few calls to getGroovyClass with calls to getLispClass, we've done most of the necessary work. For various reasons, Portofino needs to check whether a certain action class is user-defined or not. Groovy classes all implement an interface, GroovyObject, so the system can test for it to detect classes written in Groovy. ABCL doesn't add any marker interface by itself, so we'll have to define our own (say, an empty Lisp/Action interface) and remember to use it when we define our action classes. Also, since ABCL's startup time is noticeable, we can add a little piece of code – not shown here – to load it at startup rather than at the first invocation of an action.

We are now ready to replace our Groovy action. Portofino comes with a simple demo ticket-tracker application based on the database of Redmine, a popular Ruby based issue tracker. This built-in application allows to manage Projects and Issues and shows a few charts for added eye candy. We'll take as an example a modified version of the action that powers the Projects page: a class that customizes the standard CRUD page to add computed values for the project's create and update date, and to add a shortcut button to immediately show the form for creating a new ticket for that project. It is important to note that objects manipulated by CRUD pages are Java maps, for which Groovy has built-in syntax to access their contents, but which in Java would require explicit put() and get() method calls.

```
@SupportPermissions(...)  
@RequiresPermissions(level = AccessLevel.VIEW)
```
focusing on the business logic is a desirable goal, and Lisp offers that limiting the exposure to implementation details and people to develop relatively complex applications, it appears Lisp, if we shift our perspective a little. Considering that one of Java – is also in our opinion its weakest point compared to Groovy's strongest point – that it feels just like a dynamic version persistent objects in Portofino easier than in Java.

Convenient syntax for accessing maps makes dealing with to provide autocomplete functionality, how to debug it. Java. Popular Java IDEs know how to colorize Groovy code, how programmer. Code is naturally organized in classes just like in syntax is mostly familiar (with a few warts) to every Java Groovy has a few strong points for this particular use case. Its

6. REFERENCES

[18] http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html
[22] http://docs.codehaus.org/display/GROOVY/Writing+Domain-Specific+Languages