# A Manual for Armed Bear Common Lisp

Mark Evenson, Erik Huelsmann, Alessio Stalla, Ville Voutilainen

October 21, 2011

# Chapter 1

# Introduction

Armed Bear is a (mostly) conforming implementation of the ANSI Common Lisp standard. This manual documents the Armed Bear Common Lisp implementation for users of the system.

### 1.0.1 Version

This manual corresponds to abcl-1.0.0, released on October 22, 2011.

### 1.0.2 License



### 1.0.3 Contributors

Philipp Marek
Douglas Miles
Alan Ruttenberg
and of course
Peter Graves

# Chapter 2

# Running

ABCL is packaged as a single jar file usually named either "abcl.jar" or possibly"abcl-1.0.0.jar" if one is using a versioned package from your system vendor. This byte archive can be executed under the control of a suitable JVM by using the "-jar" option to parse the manifest, and select the named class ( `org.armedbear.lisp.Main` ) for execution, viz:

```
cmd$ java -jar abcl.jar
```

N.b. for the proceeding command to work, the "java" executable needs to be in your path.

To make it easier to facilitate the use of ABCL in tool chains (such as SLIME [1]) the invocation is wrapped in a Bourne shell script under UNIX or a DOS command script under Windows so that ABCL may be executed simply as:

```
cmd$ abcl
```

## 2.1 Options

ABCL supports the following command line options:

```
--help
    Displays this message.
--noinform
    Suppresses the printing of startup information and banner.
--noinit
    Suppresses the loading of the '~/.abclrc' startup file.
--nosystem
    Suppresses loading the 'system.lisp' customization file.
--eval <FORM>
    Evaluates the <FORM> before initializing REPL.
--load <FILE>
    Loads the file <FILE> before initializing REPL.
--load-system-file <FILE>
    Loads the system file <FILE> before initializing REPL.
--batch
    The process evaluates forms specified by arguments and possibly by those
    by those in the intialization file '~/.abcl', and then exits.

The occurance of '--' copies the remaining arguments, unprocessed, into
the variable EXTENSIONS:*COMMAND-LINE-ARGUMENT-LIST*.
```

[1] SLIME is the Superior Lisp Mode for Interaction under Emacs

All of the command line arguments which follow the occurrence of "—-" are passed into a list bound to the EXT:*COMMAND-LINE-ARGUMENT-LIST* variable.

## 2.2 Initialization

If the ABCL process is started without the "–noinit" flag, it attempts to load a file named ".abclrc" located in the user's home directory and then interpret its contents.

The user's home directory is determined by the value of the JVM system property "user.home". This value may–or may not–correspond to the value of the HOME system environment variable at the discretion of the JVM implementation that ABCL finds itself hosted upon.

# Chapter 3

# Conformance

## 3.1 ANSI Common Lisp

ABCL is currently a (non)-conforming ANSI Common Lisp implementation due to the following known issues:

- The generic function signatures of the DOCUMENTATION symbol do not match the CLHS.
- The TIME form does not return a proper VALUES environment to its caller.

Somewhat confusingly, this statement of non-conformance in the accompanying user documentation fullfills the requirements that ABCL is a conforming ANSI Common Lisp implementation according to the CLHS [1]. Clarifications to this point are solicited.

ABCL aims to be be a fully conforming ANSI Common Lisp implementation. Any other behavior should be reported as a bug.

## 3.2 Contemporary Common Lisp

In addition to ANSI conformance, ABCL strives to implement features expected of a contemporary Common Lisp [2]

### 3.2.1 Deficiencies

The following known problems detract from ABCL being a proper contemporary Comon Lisp.

- An incomplete implementation of a properly named metaobject protocol (viz. (A)MOP [3] )
- Incomplete streams abstraction, in that ABCL needs suitable abstraction between ANSI and Gray streams. The streams could be optimized to the JVM NIO abstractions at great profit for binary byte-level manipulations.
- Incomplete documentation (missing docstrings from exported symbols.

[1] Common Lisp Hyperspec language reference document.
[2] i.e. a Lisp of the post 2005 Renaissance
[3] Another Metaobject Protocol

# Chapter 4

# Interaction with Hosting JVM

The Armed Bear Common Lisp implementation is hosted on a Java Virtual Machine. This chapter describes the mechanisms by which the implementation interacts with that hosting mechanism.

## 4.1 Lisp to Java

ABCL offers a number of mechanisms to interact with Java from its Lisp environment. It allows calling both instance and static methods of Java objects, manipulation of instance and static fields on Java objects, and construction of new Java objects.

When calling Java routines, some values will automatically be converted by the FFI [1] from Lisp values to Java values. These conversions typically apply to strings, integers and floats. Other values need to be converted to their Java equivalents by the programmer before calling the Java object method. Java values returned to Lisp are also generally converted back to their Lisp counterparts. Some operators make an exception to this rule and do not perform any conversion; those are the "raw" counterparts of certain FFI functions and are recognizable by their name ending with `-RAW` .

### 4.1.1 Low-level Java API

We define a higher level Java API in the **??**(JSS package) which is available in the `contrib/ ??` directory. This package is described later in this document. This section covers the lower level API directly available after evaluating `(require 'JAVA)` .

#### Calling Java Object Methods

There are two ways to call a Java object method in the low-level (basic) API:

- Call a specific method reference (which was previously acquired)
- Dynamic dispatch using the method name and the call-specific arguments provided by finding the **??**best match.

The dynamic dispatch variant is discussed in the next section.

`JAVA:JMETHOD` is used to acquire a specific method reference. The function takes at two or more arguments. The first is Java class designator (a `JAVA:JAVA-CLASS` object returned by `JAVA:JCLASS` or a string naming a Java class). The second is a string naming the method.

Any arguments beyond the first two should be strings naming Java classes with one exception as listed in the next paragraph. These classes specify the types of the arguments for the method to be returned.

[1] FFI stands for Foreign Function Interface which is the term of art which describes how a Lisp implementation encapsulates invocation in other languages.

There's additional calling convention to the `JAVA:JMETHOD` function: When the method is called with three parameters and the last parameter is an integer, the first method by that name and matching number of parameters is returned.

Once one has a reference to the method, one may invoke it using `JAVA:JCALL`, which takes the method as the first argument. The second argument is the object instance to call the method on, or `NIL` in case of a static method. Any remaining parameters are used as the remaining arguments for the call.

### Calling Java object methods: dynamic dispatch

The second way of calling Java object methods is by using dynamic dispatch. In this case `JAVA:JCALL` is used directly without acquiring a method reference first. In this case, the first argument provided to `JAVA:JCALL` is a string naming the method to be called. The second argument is the instance on which the method should be called and any further arguments are used to select the best matching method and dispatch the call.

### Dynamic dispatch: Caveats

Dynamic dispatch is performed by using the Java reflection API [2]. Generally the dispatch works fine, but there are corner cases where the API does not correctly reflect all the details involved in calling a Java method. An example is the following Java code:

```
ZipFile jar = new ZipFile("/path/to/some.jar");
Object els = jar.entries();
Method method = els.getClass().getMethod("hasMoreElements");
method.invoke(els);
```

even though the method `hasMoreElements()` is public in `Enumeration`, the above code fails with

```
java.lang.IllegalAccessException: Class ... can
not access a member of class java.util.zip.ZipFile\$2 with modifiers
"public"
        at sun.reflect.Reflection.ensureMemberAccess(Reflection.java:65)
        at java.lang.reflect.Method.invoke(Method.java:583)
        at ...
```

because the method has been overridden by a non-public class and the reflection API, unlike javac, is not able to handle such a case.

While code like that is uncommon in Java, it is typical of ABCL's FFI calls. The code above corresponds to the following Lisp code:

```
(let ((jar (jnew "java.util.zip.ZipFile" "/path/to/some.jar")))
  (let ((els (jcall "entries" jar)))
    (jcall "hasMoreElements" els)))
```

except that the dynamic dispatch part is not shown.

To avoid such pitfalls, all Java objects in ABCL carry an extra field representing the "intended class" of the object. That is the class that is used first by `JAVA:JCALL` and similar to resolve methods; the actual class of the object is only tried if the method is not found in the intended class. Of course, the intended class is always a super-class of the actual class - in the worst case, they coincide. The intended class is deduced by the return type of the method that originally returned the Java object; in the case above, the intended class of `ELS` is `java.util.Enumeration` because that's the return type of the `entries` method.

While this strategy is generally effective, there are cases where the intended class becomes too broad to be useful. The typical example is the extraction of an element from a collection, since

[2] The Java reflection API is found in the `java.lang.reflect` package

methods in the collection API erase all types to `Object` . The user can always force a more specific intended class by using the `JAVA:JCOERCE` operator.

#### Calling Java class static methods

Like with non-static methods, references to static methods can be acquired by using the `JAVA:JMETHOD` primitive. In order to call this method, it's not possible to use the `JAVA:JCALL` primitive however: there's a separate API to retrieve a reference to static methods. This primitive is called `JAVA:JSTATIC` .

Like `JAVA:JCALL` , `JAVA:JSTATIC` supports dynamic dispatch by passing the name of the method as a string instead of passing a method reference. The parameter values should be values to pass in the function call instead of a specification of classes for each parameter.

#### Parameter matching for FFI dynamic dispatch

The algorithm used to resolve the best matching method given the name and the arguments' types is the same as described in the Java Language Specification. Any deviation should be reported as a bug.

#### Instantiating Java objects

Java objects can be instantiated (created) from Lisp by calling a constructor from the class of the object to be created. The same way `JAVA:JMETHOD` is used to acquire a method reference, the `JAVA:JCONSTRUCTOR` primitive can be used to acquire a constructor reference. It's arguments specify the types of arguments of the constructor method the same way as with `JAVA:JMETHOD` .

The constructor can't be passed to `JAVA:JCALL` , but instead should be passed as an argument to `JAVA:JNEW` .

#### Accessing Java object fields

Fields in Java objects can be accessed using the getter and setter functions `JAVA:GETFIELD` and `JAVA:PUTFIELD` . This applies to values stored in object instances. If you want to access static fields: see the next section.

Like `JAVA:JCALL` and friends, values returned from these accessors carry an intended class around and values which can be converted to Lisp values will be converted.

#### Accessing Java static fields

Static fields in Java objects (class fields) can be accessed using the getter and setter functions `JAVA:GETSTATIC` and `JAVA:PUTSTATIC` . Values stored in object instance fields can be accessed as described in the previous section.

Like `JAVA:JCALL` and friends, values returned from these accessors carry an intended class around and values which can be converted to Lisp values will be converted.

## 4.2 Lisp from Java

In order to access the Lisp world from Java, one needs to be aware of a few things. The most important ones are listed below.

- All Lisp values are descendants of LispObject.java
- In order to
- Lisp symbols are accessible via either directly referencing the Symbol.java instance or by dynamically introspecting the corresponding Package.java instance.

- The Lisp dynamic environment may be saved via `LispThread.bindSpecial(Binding)` and restored via `LispThread.resetSpecialBindings(Mark)` .
- Functions may be executed by invocation of the Function.execute(args [...])

### 4.2.1 Lisp FFI

FFI stands for "Foreign Function Interface" which is the phase which the contemporary Lisp world refers to methods of "calling out" from Lisp into "foreign" languages and environments. This document describes the various ways that one interacts with Lisp world of ABCL from Java, considering the hosted Lisp as the "Foreign Function" that needs to be "Interfaced".

### 4.2.2 Calling Lisp from Java

Note: As the entire ABCL Lisp system resides in the org.armedbear.lisp package the following code snippets do not show the relevant import statements in the interest of brevity. An example of the import statement would be

```
import org.armedbear.lisp.*;
```

to potentially import all the JVM symbol from the 'org.armedbear.lisp' namespace.

Per JVM, there can only ever be a single Lisp interpreter. This is started by calling the static method 'Interpreter.createInstance()'.

```
Interpreter interpreter = Interpreter.createInstance();
```

If this method has already been invoked in the lifetime of the current Java process it will return null, so if you are writing Java whose life-cycle is a bit out of your control (like in a Java servlet), a safer invocation pattern might be:

```
Interpreter interpreter = Interpreter.getInstance();
if (interpreter == null) {
  interpreter = Interpreter.createInstance();
}
```

The Lisp `eval` primitive may be simply passed strings for evaluation, as follows

```
String line = "(load \"file.lisp\")";
LispObject result = interpreter.eval(line);
```

Notice that all possible return values from an arbitrary Lisp computation are collapsed into a single return value. Doing useful further computation on the "LispObject" depends on knowing what the result of the computation might be, usually involves some amount of `instanceof` introspection, and forms a whole topic to itself (c.f. [Introspecting a LispObject])

Using `eval` involves the Lisp interpreter. Lisp functions may be directly invoked by Java method calls as follows. One simply locates the package containing the symbol, then obtains a reference to the symbol, and then invokes the `execute()` method with the desired parameters.

```
interpreter.eval("(defun foo (msg) (format nil \"You told me: '~A'~%\" msg))");
Package pkg = Packages.findPackage("CL-USER");
Symbol foo = pkg.findAccessibleSymbol("FOO");
Function fooFunction = (Function)foo.getSymbolFunction();
JavaObject parameter = new JavaObject("Lisp is fun!");
LispObject result = fooFunction.execute(parameter);
// How to get the "naked string value"?
System.out.println("The result was " + result.writeToString());
```

If one is calling an primitive function in the CL package the syntax becomes considerably simpler. If we can locate the instance of definition in the ABCL Java source, we can invoke the symbol directly. For instance, to tell if a 'LispObject' contains a reference to a symbol.

```
boolean nullp(LispObject object) {
  LispObject result = Primitives.NULL.execute(object);
  if (result == NIL) { // the symbol 'NIL' is explicitly named in the Java
                       // namespace at ''Symbol.NIL''
                       // but is always present in the
                       // local namespace in its unadorned form for
                       // the convenience of the User.
    return false;
  }
  return true;
}
```

#### Introspecting a LispObject

We present various patterns for introspecting an an arbitrary 'LispObject' which can represent the result of every Lisp evaluation into semantics that Java can meaningfully deal with.

#### LispObject as `boolean`

If the LispObject a generalized boolean values, one can use `getBooleanValue()` to convert to Java:

```
LispObject object = Symbol.NIL;
boolean javaValue = object.getBooleanValue();
```

Although since in Lisp, any value other than NIL means "true" (so-called generalized Boolean), the use of Java equality it quite a bit easier to type and more optimal in terms of information it conveys to the compiler would be:

```
boolean javaValue = (object != Symbol.NIL);
```

**LispObject is a list** If LispObject is a list, it will have the type 'Cons'. One can then use the `copyToArray` to make things a bit more suitable for Java iteration.

```
LispObject result = interpreter.eval("'(1 2 4 5)");
if (result instanceof Cons) {
  LispObject array[] = ((Cons)result.copyToArray());
  ...
}
```

A more Lispy way to iterated down a list is to use the 'cdr()' access function just as like one would traverse a list in Lisp:;

```
LispObject result = interpreter.eval("'(1 2 4 5)");
while (result != Symbol.NIL) {
  doSomething(result.car());
  result = result.cdr();
}
```

### 4.2.3 Java Scripting API (JSR-223)

ABCL can be built with support for JSR-223, which offers a language-agnostic API to invoke other languages from Java. The binary distribution downloadable from ABCL's common-lisp.net home is built with JSR-223 support. If you're building ABCL from source on a pre-1.6 JVM, you need to have a JSR-223 implementation in your CLASSPATH (such as Apache Commons BSF

3.x or greater) in order to build ABCL with JSR-223 support; otherwise, this feature will not be built.

This section describes the design decisions behind the ABCL JSR-223 support. It is not a description of what JSR-223 is or a tutorial on how to use it. See http://trac.common-lisp.net/armedbear/browser/trunk/abcl/examples/jsr-223 for example usage.

### Conversions

In general, ABCL's implementation of the JSR-223 API performs implicit conversion from Java objects to Lisp objects when invoking Lisp from Java, and the opposite when returning values from Java to Lisp. This potentially reduces coupling between user code and ABCL. To avoid such conversions, wrap the relevant objects in `JavaObject` instances.

### Implemented JSR-223 interfaces

JSR-223 defines three main interfaces, of which two (Invocable and Compilable) are optional. ABCL implements all the three interfaces - ScriptEngine and the two optional ones - almost completely. While the JSR-223 API is not specific to a single scripting language, it was designed with languages with a more or less Java-like object model in mind: languages such as Javascript, Python, Ruby, which have a concept of "class" or "object" with "fields" and "methods". Lisp is a bit different, so certain adaptations were made, and in one case a method has been left unimplemented since it does not map at all to Lisp.

### The ScriptEngine

The main interface defined by JSR-223, javax.script.ScriptEngine, is implemented by the class `org.armedbear.lisp.scripting.AbclScriptEngine` . AbclScriptEngine is a singleton, reflecting the fact that ABCL is a singleton as well. You can obtain an instance of AbclScriptEngine using the AbclScriptEngineFactory or by using the service provider mechanism through ScriptEngineManager (refer to the javax.script documentation).

### Startup and configuration file

At startup (i.e. when its constructor is invoked, as part of the static initialization phase of AbclScriptEngineFactory) the ABCL script engine attempts to load an "init file" from the classpath (/abcl-script-config.lisp). If present, this file can be used to customize the behaviour of the engine, by setting a number of variables in the ABCL-SCRIPT package. Here is a list of the available variables:

- *use-throwing-debugger* Controls whether ABCL uses a non-standard debugging hook function to throw a Java exception instead of dropping into the debugger in case of unhandled error conditions.
  - Default value: T
  - Rationale: it is more convenient for Java programmers using Lisp as a scripting language to have it return exceptions to Java instead of handling them in the Lisp world.
  - Known Issues: the non-standard debugger hook has been reported to misbehave in certain circumstances, so consider disabling it if it doesn't work for you.
- *launch-swank-at-startup* If true, Swank will be launched at startup. See *swank-dir* and *swank-port*.
  - Default value: NIL
- *swank-dir* The directory where Swank is installed. Must be set if *launch-swank-at-startup* is true.

- *swank-port* The port where Swank will listen for connections. Must be set if *launch-swank-at-startup* is true.
  - Default value: 4005

Additionally, at startup the AbclScriptEngine will `(require 'asdf)` - in fact, it uses asdf to load Swank.

### Evaluation

Code is read and evaluated in the package ABCL-SCRIPT-USER. This packages USEs the COMMON-LISP, JAVA and ABCL-SCRIPT packages. Future versions of the script engine might make this default package configurable. The `CL:LOAD` function is used under the hood for evaluating code, and thus the same behavior of LOAD is guaranteed. This allows, among other things, `IN-PACKAGE` forms to change the package in which the loaded code is read.

It is possible to evaluate code in what JSR-223 calls a "ScriptContext" (basically a flat environment of name-¿value pairs). This context is used to establish special bindings for all the variables defined in it; since variable names are strings from Java's point of view, they are first interned using READ-FROM-STRING with, as usual, ABCL-SCRIPT-USER as the default package. Variables are declared special because CL's `LOAD` , `EVAL` and `COMPILE` functions work in a null lexical environment and would ignore non-special bindings.

Contrary to what the function `LOAD` does, evaluation of a series of forms returns the value of the last form instead of T, so the evaluation of short scripts does the Right Thing.

### Compilation

AbclScriptEngine implements the `javax.script.Compilable` interface. Currently it only supports compilation using temporary files. Compiled code, returned as an instance of javax.script.CompiledScript, is read, compiled and executed by default in the ABCL-SCRIPT-USER package, just like evaluated code. Differently from evaluated code, though, due to the way the ABCL compiler works, compiled code contains no reference to top-level self-evaluating objects (like numbers or strings). Thus, when evaluated, a piece of compiled code will return the value of the last non-self-evaluating form: for example the code "(do-something) 42" will return 42 when interpreted, but will return the result of (do-something) when compiled and later evaluated. To ensure consistency of behavior between interpreted and compiled code, make sure the last form is always a compound form - at least (identity some-literal-object). Note that this issue should not matter in real code, where it is unlikely a top-level self-evaluating form will appear as the last form in a file (in fact, the Common Lisp load function always returns T upon success; with JSR-223 this policy has been changed to make evaluation of small code snippets work as intended).

### Invocation of functions and methods

AbclScriptEngine implements the `javax.script.Invocable` interface, which allows to directly call Lisp functions and methods, and to obtain Lisp implementations of Java interfaces. This is only partially possible with Lisp since it has functions, but not methods - not in the traditional OO sense, at least, since Lisp methods are not attached to objects but belong to generic functions. Thus, the method `invokeMethod()` is not implemented and throws an UnsupportedOperationException when called. The `invokeFunction()` method should be used to call both regular and generic functions.

### Implementation of Java interfaces in Lisp

ABCL can use the Java reflection-based proxy feature to implement Java interfaces in Lisp. It has several built-in ways to implement an interface, and supports definition of new ones. The `JAVA:JMAKE-PROXY` generic function is used to make such proxies. It has the following signature:

`jmake-proxy interface implementation &optional lisp-this ==> proxy`

`interface` is a Java interface metaobject (e.g. obtained by invoking `jclass` ) or a string naming a Java interface. `implementation` is the object used to implement the interface - several built-in methods of jmake-proxy exist for various types of implementations. `lisp-this` is an object passed to the closures implementing the Lisp "methods" of the interface, and defaults to `NIL` .

The returned proxy is an instance of the interface, with methods implemented with Lisp functions.

Built-in interface-implementation types include:

- a single Lisp function which upon invocation of any method in the interface will be passed the method name, the Lisp-this object, and all the parameters. Useful for interfaces with a single method, or to implement custom interface-implementation strategies.
- a hash-map of method-name -¿ Lisp function mappings. Function signature is `(lisp-this &rest args)` .
- a Lisp package. The name of the Java method to invoke is first transformed in an idiomatic Lisp name ( `javaMethodName` becomes `JAVA-METHOD-NAME` ) and a symbol with that name is searched in the package. If it exists and is fbound, the corresponding function will be called. Function signature is as the hash-table case.

This functionality is exposed by the AbclScriptEngine with the two methods getInterface(Class) and getInterface(Object, Class). The former returns an interface implemented with the current Lisp package, the latter allows the programmer to pass an interface-implementation object which will in turn be passed to the jmake-proxy generic function.

# Chapter 5

# Implementation Dependent Extensions

As outlined by the CLHS ANSI conformance guidelines, we document the extensions to the Armed Bear Lisp implementation made accessible to the user by virtue of being an exported symbol in the JAVA, THREADS, or EXTENSIONS packages.

## 5.1 JAVA

### 5.1.1 Modifying the JVM CLASSPATH

The JAVA:ADD-TO-CLASSPATH generic functions allows one to add the specified pathname or list of pathnames to the current classpath used by ABCL, allowing the dynamic loading of JVM objects:

```
CL-USER> (add-to-classpath "/path/to/some.jar")
```

N.b `add-to-classpath` only affects the classloader used by ABCL (the value of the special variable `JAVA:*CLASSLOADER*` . It has no effect on Java code outside ABCL.

### 5.1.2 API

```
%JGET-PROPERTY-VALUE
  Function: Gets a JavaBeans property on JAVA-OBJECT.
%JSET-PROPERTY-VALUE
  Function: Sets a JavaBean property on JAVA-OBJECT.
*JAVA-OBJECT-TO-STRING-LENGTH*
  Variable: Length to truncate toString() PRINT-OBJECT output for an
 otherwise unspecialized JAVA-OBJECT.  Can be set to NIL to indicate
 no limit.
+FALSE+
  Variable: The JVM primitive value for boolean false.
+NULL+
  Variable: The JVM null object reference.
+TRUE+
  Variable: The JVM primitive value for boolean true.
ADD-TO-CLASSPATH
  Generic Function: (not documented)
CHAIN
  Macro: (not documented)
DESCRIBE-JAVA-OBJECT
  Function: (not documented)
DUMP-CLASSPATH
  Function: (not documented)
ENSURE-JAVA-CLASS
  Function: (not documented)
ENSURE-JAVA-OBJECT
  Function: Ensures OBJ is wrapped in a JAVA-OBJECT, wrapping it if necessary.
GET-CURRENT-CLASSLOADER
  Function: (not documented)
GET-DEFAULT-CLASSLOADER
  Function: (not documented)
JARRAY-COMPONENT-TYPE
  Function: Returns the component type of the array type ATYPE
JARRAY-LENGTH
  Function: (not documented)
JARRAY-REF
  Function: Dereferences the Java array JAVA-ARRAY using the given
 INDICIES, coercing the result into a Lisp object, if possible.
JARRAY-REF-RAW
  Function: Dereference the Java array JAVA-ARRAY using the given
 INDICIES. Does not attempt to coerce the result into a Lisp object.
JARRAY-SET
  Function: Stores NEW-VALUE at the given index in JAVA-ARRAY.
JAVA-CLASS
  Class: (not documented)
JAVA-EXCEPTION
  Class: (not documented)
JAVA-EXCEPTION-CAUSE
  Function: (not documented)
JAVA-OBJECT
  Class: (not documented)
JAVA-OBJECT-P
  Function: Returns T if OBJECT is a JAVA-OBJECT.
JCALL
  Function: Invokes the Java method METHOD-REF on INSTANCE with
```

```
 arguments ARGS, coercing the result into a Lisp object, if possible.
JCALL-RAW
  Function: Invokes the Java method METHOD-REF on INSTANCE with
 arguments ARGS. Does not attempt to coerce the result into a Lisp
 object.
JCLASS
  Function: Returns a reference to the Java class designated by
 NAME-OR-CLASS-REF. If the CLASS-LOADER parameter is passed, the class
 is resolved with respect to the given ClassLoader.
JCLASS-ARRAY-P
  Function: Returns T if CLASS is an array class
JCLASS-CONSTRUCTORS
  Function: Returns a vector of constructors for CLASS
JCLASS-FIELD
  Function: Returns the field named FIELD-NAME of CLASS
JCLASS-FIELDS
  Function: Returns a vector of all (or just the declared/public, if
 DECLARED/PUBLIC is true) fields of CLASS
JCLASS-INTERFACE-P
  Function: Returns T if CLASS is an interface
JCLASS-INTERFACES
  Function: Returns the vector of interfaces of CLASS
JCLASS-METHODS
  Function: Return a vector of all (or just the declared/public, if
 DECLARED/PUBLIC is true) methods of CLASS
JCLASS-NAME
  Function: (not documented)
JCLASS-OF
  Function: (not documented)
JCLASS-SUPERCLASS
  Function: Returns the superclass of CLASS, or NIL if it hasn't got one
JCLASS-SUPERCLASS-P
  Function: Returns T if CLASS-1 is a superclass or interface of CLASS-2
JCOERCE
  Function: Attempts to coerce OBJECT into a JavaObject of class
 INTENDED-CLASS.  Raises a TYPE-ERROR if no conversion is possible.
JCONSTRUCTOR
  Function: Returns a reference to the Java constructor of CLASS-REF
 with the given PARAMETER-CLASS-REFS.
JCONSTRUCTOR-PARAMS
  Function: Returns a vector of parameter types (Java classes) for CONSTRUCTOR
JEQUAL
  Function: Compares obj1 with obj2 using java.lang.Object.equals()
JFIELD
  Function: Retrieves or modifies a field in a Java class or instance.
JFIELD-NAME
  Function: Returns the name of FIELD as a Lisp string
JFIELD-RAW
  Function: Retrieves or modifies a field in a Java class or instance. Does not
JFIELD-TYPE
  Function: Returns the type (Java class) of FIELD
JINSTANCE-OF-P
  Function: OBJ is an instance of CLASS (or one of its subclasses)
JINTERFACE-IMPLEMENTATION
```

```
  Function: Creates and returns an implementation of a Java interface with
JMAKE-INVOCATION-HANDLER
  Function: (not documented)
JMAKE-PROXY
  Generic Function: (not documented)
JMEMBER-PROTECTED-P
  Function: MEMBER is a protected member of its declaring class
JMEMBER-PUBLIC-P
  Function: MEMBER is a public member of its declaring class
JMEMBER-STATIC-P
  Function: MEMBER is a static member of its declaring class
JMETHOD
  Function: Returns a reference to the Java method METHOD-NAME of
 CLASS-REF with the given PARAMETER-CLASS-REFS.
JMETHOD-LET
  Macro: (not documented)
JMETHOD-NAME
  Function: Returns the name of METHOD as a Lisp string
JMETHOD-PARAMS
  Function: Returns a vector of parameter types (Java classes) for METHOD
JMETHOD-RETURN-TYPE
  Function: Returns the result type (Java class) of the METHOD
JNEW
  Function: Invokes the Java constructor CONSTRUCTOR with the arguments ARGS.
JNEW-ARRAY
  Function: Creates a new Java array of type ELEMENT-TYPE, with the given DIMENSIONS.
JNEW-ARRAY-FROM-ARRAY
  Function: Returns a new Java array with base type ELEMENT-TYPE (a string or a class-ref)
JNEW-ARRAY-FROM-LIST
  Function: (not documented)
JNEW-RUNTIME-CLASS
  Function: (not documented)
JNULL-REF-P
  Function: Returns a non-NIL value when the JAVA-OBJECT `object` is `null`,
JOBJECT-CLASS
  Function: Returns the Java class that OBJ belongs to
JOBJECT-LISP-VALUE
  Function: Attempts to coerce JAVA-OBJECT into a Lisp object.
JPROPERTY-VALUE
  Function: (not documented)
JREDEFINE-METHOD
  Function: (not documented)
JREGISTER-HANDLER
  Function: (not documented)
JRESOLVE-METHOD
  Function: Finds the most specific Java method METHOD-NAME on
 INSTANCE applicable to arguments ARGS. Returns NIL if no suitable
 method is found. The algorithm used for resolution is the same used
 by JCALL when it is called with a string as the first parameter
 (METHOD-REF).
JRUN-EXCEPTION-PROTECTED
  Function: Invokes the function CLOSURE and returns the result.
  Signals an error if stack or heap exhaustion occurs.
JRUNTIME-CLASS-EXISTS-P
```

```
  Function: (not documented)
JSTATIC
  Function: Invokes the static method METHOD on class CLASS with ARGS.
JSTATIC-RAW
  Function: Invokes the static method METHOD on class CLASS with
 ARGS. Does not attempt to coerce the arguments or result into a Lisp
 object.
MAKE-CLASSLOADER
  Function: (not documented)
MAKE-IMMEDIATE-OBJECT
  Function: Attempts to coerce a given Lisp object into a java-object of the
REGISTER-JAVA-EXCEPTION
  Function: Registers the Java Throwable named by the symbol
 EXCEPTION-NAME as the condition designated by CONDITION-SYMBOL.
  Returns T if successful, NIL if not.
UNREGISTER-JAVA-EXCEPTION
  Function: Unregisters the Java Throwable EXCEPTION-NAME previously
 registered by REGISTER-JAVA-EXCEPTION.
```

## 5.2 THREADS

The extensions for handling multithreaded execution are collected in the `THREADS` package. Most of the abstractions in Doug Lea's excellent `java.util.concurrent` packages may be manipulated directly via the JSS contrib to great effect.

### 5.2.1 API

```
THREADS:CURRENT-THREAD
  Function: (not documented)
THREADS:DESTROY-THREAD
  Function: (not documented)
THREADS:GET-MUTEX
  Function: Acquires a lock on the ‘mutex’.
THREADS:INTERRUPT-THREAD
  Function: Interrupts THREAD and forces it to apply FUNCTION to ARGS.
THREADS:MAILBOX-EMPTY-P
  Function: Returns non-NIL if the mailbox can be read from, NIL otherwise.
THREADS:MAILBOX-PEEK
  Function: Returns two values. The second returns non-NIL when the mailbox
THREADS:MAILBOX-READ
  Function: Blocks on the mailbox until an item is available for reading.
THREADS:MAILBOX-SEND
  Function: Sends an item into the mailbox, notifying 1 waiter
THREADS:MAKE-MAILBOX
  Function: (not documented)
THREADS:MAKE-MUTEX
  Function: (not documented)
THREADS:MAKE-THREAD
  Function: (not documented)
THREADS:MAKE-THREAD-LOCK
  Function: Returns an object to be used with the ‘with-thread-lock’ macro.
THREADS:MAPCAR-THREADS
  Function: (not documented)
THREADS:OBJECT-NOTIFY
  Function: (not documented)
THREADS:OBJECT-NOTIFY-ALL
  Function: (not documented)
THREADS:OBJECT-WAIT
  Function: (not documented)
THREADS:RELEASE-MUTEX
  Function: Releases a lock on the ‘mutex’.
THREADS:SYNCHRONIZED-ON
  Function: (not documented)
THREADS:THREAD
  Class: (not documented)
THREADS:THREAD-ALIVE-P
  Function: Boolean predicate whether THREAD is alive.
THREADS:THREAD-JOIN
  Function: Waits for thread to finish.
THREADS:THREAD-NAME
  Function: (not documented)
THREADS:THREADP
  Function: (not documented)
THREADS:WITH-MUTEX
  Function: (not documented)
THREADS:WITH-THREAD-LOCK
  Function: (not documented)
```

## 5.3 EXTENSIONS

The symbols in the EXTENSIONS package (nicknamed "EXT") constitutes extensions to the ANSI standard that are potentially useful to the user. They include functions for manipulating network sockets, running external programs, registering object finalizers, constructing reference weakly held by the garbage collector and others.

See **??** for a generic function interface to the native JVM contract for `java.util.List` .

### 5.3.1 API

```
%CADDR
  Macro: (not documented)
%CADR
  Macro: (not documented)
%CAR
  Macro: (not documented)
%CDR
  Macro: (not documented)
*AUTOLOAD-VERBOSE*
  Variable: (not documented)
*BATCH-MODE*
  Variable: (not documented)
*COMMAND-LINE-ARGUMENT-LIST*
  Variable: (not documented)
*DEBUG-CONDITION*
  Variable: (not documented)
*DEBUG-LEVEL*
  Variable: (not documented)
*DISASSEMBLER*
  Variable: (not documented)
*ED-FUNCTIONS*
  Variable: (not documented)
*ENABLE-INLINE-EXPANSION*
  Variable: (not documented)
*INSPECTOR-HOOK*
  Variable: (not documented)
*LISP-HOME*
  Variable: (not documented)
*LOAD-TRUENAME-FASL*
  Variable: (not documented)
*PRINT-STRUCTURE*
  Variable: (not documented)
*REQUIRE-STACK-FRAME*
  Variable: (not documented)
*SAVED-BACKTRACE*
  Variable: (not documented)
*SUPPRESS-COMPILER-WARNINGS*
  Variable: (not documented)
*WARN-ON-REDEFINITION*
  Variable: (not documented)
ADJOIN-EQL
  Function: (not documented)
ARGLIST
  Function: (not documented)
ASSQ
  Function: (not documented)
ASSQL
  Function: (not documented)
AUTOLOAD
  Function: (not documented)
AUTOLOAD-MACRO
  Function: (not documented)
AUTOLOADP
  Function: (not documented)
```

```
AVER
  Macro: (not documented)
CANCEL-FINALIZATION
  Function: (not documented)
CHAR-TO-UTF8
  Function: (not documented)
CHARPOS
  Function: (not documented)
CLASSP
  Function: (not documented)
COLLECT
  Macro: (not documented)
COMPILE-FILE-IF-NEEDED
  Function: (not documented)
COMPILE-SYSTEM
  Function: (not documented)
COMPILER-ERROR
  Function: (not documented)
  Class: (not documented)
COMPILER-UNSUPPORTED-FEATURE-ERROR
  Class: (not documented)
DESCRIBE-COMPILER-POLICY
  Function: (not documented)
DOUBLE-FLOAT-NEGATIVE-INFINITY
  Variable: (not documented)
DOUBLE-FLOAT-POSITIVE-INFINITY
  Variable: (not documented)
DUMP-JAVA-STACK
  Function: (not documented)
EXIT
  Function: (not documented)
FEATUREP
  Function: (not documented)
FILE-DIRECTORY-P
  Function: (not documented)
FINALIZE
  Function: (not documented)
FIXNUMP
  Function: (not documented)
GC
  Function: (not documented)
GET-FLOATING-POINT-MODES
  Function: (not documented)
GET-SOCKET-STREAM
  Function: :ELEMENT-TYPE must be CHARACTER or (UNSIGNED-BYTE 8); the default is CHARACTER.
GETENV
  Function: Return the value of the environment VARIABLE if it exists, otherwise return NIL.
GROVEL-JAVA-DEFINITIONS
  Function: (not documented)
INIT-GUI
  Function: (not documented)
INTERNAL-COMPILER-ERROR
  Function: (not documented)
  Class: (not documented)
```

```
INTERRUPT-LISP
  Function: (not documented)
JAR-PATHNAME
  Class: (not documented)
MACROEXPAND-ALL
  Function: (not documented)
MAILBOX
  Class: (not documented)
MAKE-DIALOG-PROMPT-STREAM
  Function: (not documented)
MAKE-SERVER-SOCKET
  Function: (not documented)
MAKE-SLIME-INPUT-STREAM
  Function: (not documented)
MAKE-SLIME-OUTPUT-STREAM
  Function: (not documented)
MAKE-SOCKET
  Function: (not documented)
MAKE-TEMP-FILE
  Function: (not documented)
MAKE-WEAK-REFERENCE
  Function: (not documented)
MEMQ
  Function: (not documented)
MEMQL
  Function: (not documented)
MOST-NEGATIVE-JAVA-LONG
  Variable: (not documented)
MOST-POSITIVE-JAVA-LONG
  Variable: (not documented)
MUTEX
  Class: (not documented)
NEQ
  Function: (not documented)
NIL-VECTOR
  Class: (not documented)
PATHNAME-JAR-P
  Function: (not documented)
PATHNAME-URL-P
  Function: Predicate for whether PATHNAME references a URL.
PRECOMPILE
  Function: (not documented)
PROBE-DIRECTORY
  Function: (not documented)
PROCESS
  Function: (not documented)
PROCESS-ALIVE-P
  Function: (not documented)
PROCESS-ERROR
  Function: (not documented)
PROCESS-EXIT-CODE
  Function: (not documented)
PROCESS-INPUT
  Function: (not documented)
```

```
PROCESS-KILL
  Function: (not documented)
PROCESS-OUTPUT
  Function: (not documented)
PROCESS-P
  Function: (not documented)
PROCESS-WAIT
  Function: (not documented)
QUIT
  Function: (not documented)
RESOLVE
  Function: (not documented)
RUN-PROGRAM
  Function: (not documented)
RUN-SHELL-COMMAND
  Function: (not documented)
SERVER-SOCKET-CLOSE
  Function: (not documented)
SET-FLOATING-POINT-MODES
  Function: (not documented)
SHOW-RESTARTS
  Function: (not documented)
SIMPLE-SEARCH
  Function: (not documented)
SIMPLE-STRING-FILL
  Function: (not documented)
SIMPLE-STRING-SEARCH
  Function: (not documented)
SINGLE-FLOAT-NEGATIVE-INFINITY
  Variable: (not documented)
SINGLE-FLOAT-POSITIVE-INFINITY
  Variable: (not documented)
SLIME-INPUT-STREAM
  Class: (not documented)
SLIME-OUTPUT-STREAM
  Class: (not documented)
SOCKET-ACCEPT
  Function: (not documented)
SOCKET-CLOSE
  Function: (not documented)
SOCKET-LOCAL-ADDRESS
  Function: Returns the local address of the given socket as a dotted quad string.
SOCKET-LOCAL-PORT
  Function: Returns the local port number of the given socket.
SOCKET-PEER-ADDRESS
  Function: Returns the peer address of the given socket as a dotted quad string.
SOCKET-PEER-PORT
  Function: Returns the peer port number of the given socket.
SOURCE
  Function: (not documented)
SOURCE-FILE-POSITION
  Function: (not documented)
SOURCE-PATHNAME
  Function: (not documented)
```

```
SPECIAL-VARIABLE-P
  Function: (not documented)
STRING-FIND
  Function: (not documented)
STRING-INPUT-STREAM-CURRENT
  Function: (not documented)
STRING-POSITION
  Function: (not documented)
STYLE-WARN
  Function: (not documented)
TRULY-THE
  Special Operator: (not documented)
UPTIME
  Function: (not documented)
URI-DECODE
  Function: (not documented)
URI-ENCODE
  Function: (not documented)
URL-PATHNAME
  Class: (not documented)
URL-PATHNAME-AUTHORITY
  Function: (not documented)
URL-PATHNAME-FRAGMENT
  Function: (not documented)
URL-PATHNAME-QUERY
  Function: (not documented)
URL-PATHNAME-SCHEME
  Function: (not documented)
WEAK-REFERENCE
  Class: (not documented)
WEAK-REFERENCE-VALUE
  Function: (not documented)
```

# Chapter 6

# Beyond ANSI

Naturally, in striving to be a useful contemporary Common Lisp implementation, ABCL endeavors to include extensions beyond the ANSI specification which are either widely adopted or are especially useful in working with the hosting JVM.

## 6.1 Implementation Dependent

1. Compiler to JVM 5 bytecode
2. Pathname extensions

## 6.2 Pathname

We implement an extension to the Pathname that allows for the description and retrieval of resources named in a URI scheme that the JVM "understands". Support is built-in to the "http" and "https" implementations but additional protocol handlers may be installed at runtime by having JVM symbols present in the sun.net.protocol.dynmamic pacakge. See [JAVA2006] for more details.

ABCL has created specializations of the ANSI Pathname object to enable to use of URIs to address dynamically loaded resources for the JVM. A URL-PATHNAME has a corresponding URL whose cannoical representation is defined to be the NAMESTRING of the Pathname.

```
JAR-PATHNAME isa URL-PATHNAME isa PATHNAME
```

Both URL-PATHNAME and JAR-PATHNAME may be used anywhere a PATHNAME is accepted with the following caveats:

- A stream obtained via OPEN on a URL-PATHNAME cannot be the target of write operations.
- No canonicalization is performed on the underlying URI (i.e. the implementation does not attempt to compute the current name of the representing resource unless it is requested to be resolved.) Upon resolution, any cannoicalization procedures followed in resolving the resource (e.g. following redirects) are discarded.

The implementation of URL-PATHNAME allows the ABCL user to laod dynamically code from the network. For example, for Quicklisp.

```
CL-USER> (load "http://beta.quicklisp.org/quicklisp.lisp")
```

will load and execute the Quicklisp setup code.
??

**Implementation**

DEVICE either a string denoting a drive letter under DOS or a cons specifying a URL-PATHNAME.

## 6.3 Extensible Sequences

See ?? RHODES2007 for the design.

The SEQUENCE package fully implements Christopher Rhodes' proposal for extensible sequences. These user extensible sequences are used directly in `java-collections.lisp` provide these CLOS abstractions on the standard Java collection classes as defined by the `java.util.List` contract.

This extension is not automatically loaded by the implementation. It may be loaded via:

```
CL-USER> (require 'java-collections)
```

if both extensible sequences and their application to Java collections is required, or

```
CL-USER> (require 'extensible-sequences)
```

if only the extensible sequences API as specified in ?? is required.

Note that `(require 'java-collections)` must be issued before `java.util.List` or any subclass is used as a specializer in a CLOS method definition (see the section below).

## 6.4 Extensions to CLOS

There is an additional syntax for specializing the parameter of a generic function on a java class, viz. `(java:jclass CLASS-STRING)` where `CLASS-STRING` is a string naming a Java class in dotted package form.

For instance the following specialization would perhaps allow one to print more information about the contents of a java.util.Collection object

```
(defmethod print-object ((coll (java:jclass "java.util.Collection"))
                         stream)
  ;;; ...
)
```

If the class had been loaded via a classloader other than the original the class you wish to specialize on, one needs to specify the classloader as an optional third argument.

```
(defparameter *other-classloader*
  (jcall "getBaseLoader" cl-user::*classpath-manager*))

(defmethod print-object ((device-id (java:jclass "dto.nbi.service.hdm.alcatel.com.NBIDe
                         stream)
  ;;; ...
)
```

## 6.5 Extensions to the Reader

We implement a special hexadecimal escape sequence for specifying characters to the Lisp reader, namely we allow a sequences of the form # \Uxxxx to be processed by the reader as character whose code is specified by the hexadecimal digits "xxxx". The hexadecimal sequence must be exactly four digits long [1], padded by leading zeros for values less than 0x1000.

[1] This represents a compromise with contemporary in 2011 32bit hosting architecures for which we wish to make text processing efficient. Should the User require more control over UNICODE processing we recommend Edi Weisz' excellent work with FLEXI-STREAMS which we fully support

Note that this sequence is never output by the implementation. Instead, the corresponding Unicode character is output for characters whose code is greater than 0x00ff.

### 6.5.1 JSS optionally extends the Reader

The JSS contrib consitutes an additional, optional extension to the reader in the definition of the #ïeader macro.

## 6.6 ASDF

asdf-2.017.22 is packaged as core component of ABCL, but not intialized by default, as it relies on the CLOS subsystem which can take a bit of time to initialize. It may be initialized by the ANSI REQUIRE mechanism as follows:

```
CL-USER> (require 'asdf)
```

# Chapter 7

# Contrib

## 7.1 abcl-asdf

This contrib to ABCL enables an additional syntax for ASDF system definition which dynamically loads JVM artifacts such as jar archives via a Maven encapsulation. The Maven Aether can also be directly manipulated by the function associated with the RESOLVE-DEPENDENCIES symbol.

The following ASDF components are added: JAR-FILE, JAR-DIRECTORY, CLASS-FILE-DIRECTORY and MVN.

### 7.1.1 ABCL-ASDF Examples

```
;;;; -*- Mode: LISP -*-
(in-package :asdf)

(defsystem :log4j
  :components ((:mvn "log4j/log4j"
                     :version "1.4.9")))
```

### 7.1.2 abcl-asdf API

We define an API as consisting of the following ASDF classes:

JAR-DIRECTORY, JAR-FILE, and CLASS-FILE-DIRECTORY for JVM artifacts that have a currently valid pathname representation

And the MVN and IRI classes descend from ASDF-COMPONENT, but do not directly have a filesystem location.

For use outside of ASDF, we currently define one method, RESOLVE-DEPENDENCIES which locates, downloads, caches, and then loads into the currently executing JVM process all recursive dependencies annotated in the Maven pom.xml graph.

### 7.1.3 ABCL-ASDF Example 2

Bypassing ASDF, one can directly issue requests for the Maven artifacts to be downloaded

```
CL-USER> (abcl-asdf:resolve-dependencies "com.google.gwt" "gwt-user")
WARNING: Using LATEST for unspecified version.
"/Users/evenson/.m2/repository/com/google/gwt/gwt-user/2.4.0-rc1/gwt-user-2.4.0-rc1
```

To actually load the dependency, use the JAVA:ADD-TO-CLASSPATH generic function:

```
CL-USER> (java:add-to-classpath (abcl-asdf:resolve-dependencies "com.google.gwt" "gw
```

Notice that all recursive dependencies have been located and installed locally from the network as well.

## 7.2 asdf-jar

ASDF-JAR provides a system for packaging ASDF systems into jar archives for ABCL. Given a running ABCL image with loadable ASDF systems the code in this package will recursively package all the required source and fasls in a jar archive.

## 7.3 jss

To one used to a syntax that can construct macros, the Java syntax sucks, so we introduce the #" macro.

### 7.3.1 JSS usage

Example:

```
CL-USER> (require 'jss)

CL-USER) (#"getProperties" 'java.lang.System)

CL-USER) (#"propertyNames" (#"getProperties" 'java.lang.System))
```

## 7.4 asdf-install

An implementation of ASDF-INSTALL. Superseded by Quicklisp (qv.)

# Chapter 8

# History

ABCL was originally the extension language for the J editor, which was started in 1998 by Peter Graves. Sometime in 2003, a whole lot of code that had previously not been released publically was suddenly committed that enabled ABCL to be plausibly termed an emergent ANSI Common Lisp implementation canidate.

From 2006 to 2008, Peter manned the development lists, incorporating patches as made sense. After a suitable search, Peter nominated Erik Huelsmann to take over the project.

In 2008, the implementation was transferred to the current maintainers, who have strived to improve its usability as a contemporary Common Lisp implementation.

On October 22, 2011, with the publication of this Manual explicitly stating the conformance of Armed Bear Common Lisp to ANSI, we released abcl-1.0.0.

## 8.1 References

[Java2000]: A New Era for Java Protocol Handlers. http://java.sun.com/developer/onlineTraining/protocolhandlers/

[Xach2011]: Quicklisp: A system for quickly constructing Common Lisp libraries. http://www.quicklisp.org/

[RHODES2007]: Christopher Rhodes