javahack egg

A simple interface to Java(tm)
Extension for Chicken Scheme
Version 1.0

felix

Table of Contents

1 About this egg

1.1 Version history

`1.0`	Initial release

1.2 Usage

Load this egg like so:

```
(require-extension javahack)
```

2 Documentation

This library provides a simple and convenient interface to Java. By running java as a subprocess and reading/writing s-expressions, no JNI and OS-specific hacks are needed which makes this approach more portable and robust.

On the Java side jscheme is used to parse expressions passed from CHICKEN and to convert results back into a Scheme-friendly format.

(`jscheme.jar` version 6.1 is included in this egg)

`java` [macro]

```
(java SYMBOL)
```

Returns the value represented by `SYMBOL` which should designate a Java class, field, or method, using 'Java Dot' notation:

Syntax	Type of Member	Example
"." at the end	constructor	`((java Font.) NAME STYLE SIZE)`
"." at the beginning	instance member	`((java .setFont) COMP FONT)`
"." at beginning and "$" at the end	instance field	`(define (mycar x) ((java .first$) x))` `(define (myset-car! x y) ((java .first$) x y))`
"." only in the middle	static member	`((java Math.round) 123.456)`
".class" suffix	Java class	`(java Font.class)`
"$" at the end	static field	`(java Font.BOLD$)` `(set! (java "U.useJavaSyntax$") #t)`
"$" in the middle	inner class	`(java java.awt.geom.Point2D$Double.class)`
"$" at the beginning	packageless class	`(java $ParseDemo.class)`

"#" at the end	allow private access	`((java .name$#) ((java Symbol.#) "abc"))`

Notes:

- Each evaluation of the `java` macro sends an expression to the java process and receives a result expression. Since most of the results will stay constant throughout the lifetime of the session (as they refer to classes, fields and methods), they are cached and subsequent evaluation of the same `(java ...)` expression will refer to the cached value instead. You can disable this caching (in case you are doing rather funky things) with thhe help of the `java-enable-cache` form (see below).
- Setting static fields is supported through a SRFI-17 setter, as in the example above. Since the field-name is evaluated before being sent to the java-process, pass the field as a string.
- Java objects which are returned from the Java-side can be safely used and are only garbage collected when no more references exist.

`java-enable-cache` [macro]

`(java-enable-cache FLAG)`

Enables or disables caching of results returned by the `java` macro. `FLAG` should be either the symbol `on` or `off`. Caching is enabled by default.

`java-run` [procedure]

`(java-run #!key java jar debug options classpath)`

Starts the java-VM as a subprocess, with any additional arguments customizing the JVM invocation. `java` specifies the jvm executable and defaults to `java`. `jar` gives the location of the jscheme jar file, `options` and `classpath` can be used to customize where the JVM should search for support classes and libraries, together with JVM specific options.

Passing `#t` for the `debug` parameter will print information about the message flow between CHICKEN and jscheme.

`java-stop` [procedure]

`(java-stop)`

Terminates the java process.

`java-send` [procedure]

`(java-send EXPR)`

Send an expression to Java and returns whatever result is passed back.

`java-import` [procedure]

`(java-import STRING ...)`

Imports Java packages. `STRING` should be a qualified package identifier, like `"java.lang.*"`.

`java-object?` [procedure]

```
(java-object? X)
```

Returns `#t` if `X` is a raw Java object (that can not be meaningfully converted into a Scheme value).

3 Examples

```
(use javahack)

(java-run debug: #t)

(define s (java String.class))
(pp s)
(define s1 ((java String.) "hello!"))
(pp s1)
(do ((n 2 (sub1 n))) ((zero? n))
  (pp ((java .hashCode) s1)) )
(set! s #f)
(set! s1 #f)
```

Another example, a minimal SWT application. It assumes `swt.jar` and all necessary native libraries are in the path, or in the current directory:

```
(use javahack)

(java-run debug: #t options: '("-Djava.library.path=.") classpath: "swt.jar")

(java-import "org.eclipse.swt.*")
(java-import "org.eclipse.swt.widgets.*")
(java-import "org.eclipse.swt.graphics.*")
(java-import "org.eclipse.swt.layout.*")

((java Display.setAppName) "Hello")

(define disp ((java Display.)))
(define shell ((java Shell.) disp))

((java .setLayout) shell ((java FillLayout.) (java SWT.VERTICAL$)))
((java .setText) shell "Hello, world!")

(define label ((java Label.) shell (java SWT.CENTER$)))

((java .setText) label "Hello, world")
((java .setSize) shell 300 300)
((java .open) shell)

(do () (((java .isDisposed) shell))
  (unless ((java .readAndDispatch) disp) ((java .sleep) disp)) )

((java .dispose) disp)
```

Yet another example that performs a callback. The Java code performing the callback could look like this:

```
public class Callback
```

```
{
    public static Object invoke(jsint.Procedure proc, Object x) {
        return proc.apply(new jsint.Pair(x, jsint.Pair.EMPTY));
    }
}
```

If you compile it with `javac -classpath ‘chicken-setup -repository‘/jscheme.jar Callback.java` you will have a classfile in the current directory that in combination with the following example code will call back to the CHICKEN side:

```
(use javahack)

(java-run debug: #t classpath: ".")

(write
  ((java Callback.invoke)
    (lambda (x)
      (print "ok: " x)
      42)
    "something") )
```

Index