OCaml-Java Cheat Sheet

`http://www.ocamljava.org`
Xavier Clerc, June 2015

Tools

`ocaml`	classical toplevel
`ocamlbuild`	compilation manager (`ocamljava`-aware)
`ocamlc`	compiler producing OCaml bytecode
`ocamldebug`	debugger for `ocamlc`-compiled programs
`ocamldep`	dependency analyzer
`ocamldoc`	documentation generator (`ocamljava`-aware)
`ocamlj`	toplevel using Java bytecode
`ocamljar`	post-compilation optimizer
`ocamljava`	compiler producing Java bytecode
`ocamlrun`	interpreter for `ocamlc`-compiled programs
`ocamltop`	classical toplevel, as a windowed application
`ocamlwrap`	generator of Java interfaces to OCaml code

File extensions

	`ocamlc`	`ocamlopt`	`ocamljava`
interface: source	`.mli`	`.mli`	`.mli`
compiled	`.cmi`	`.cmi`	`.cmi`
implementation: source	`.ml`	`.ml`	`.ml`
compiled	`.cmo`	`.cmx`	`.cmj`
object	-	`.o`	`.jo`
library: compiled	`.cma`	`.cmxa`	`.cmja`
object	-	`.a`	`.ja`
executable	`.out`	`.out`	`.jar`
plugin	-	`.cmxs`	`.cmjs`

Compilation and link

General

compile an interface: `ocamljava -c m.mli`
compile an implementation: `ocamljava -c m.ml`
produce a library: `ocamljava -a -o l.cmja m.cmj ...`

additional command-line switches:

`-classpath c`	set classpath
`-cp c`	add to classpath
`-java-extensions`	activate typer extensions
`-java-package p`	set package for compiled modules

Applications

link as executable: `ocamljava -o e.jar m.cmj ...`

Applets

link as applet: `ocamljava -applet k -o a.jar m.cmj ...`
where `k` is the kind of applet (`awt`, `swing`, or `graphics`)

Servlets

compile as servlet: `ocamljava -servlet k -c m.ml`
where `k` is the kind of servlet (`http`, or `generic`)
link as servlet: `ocamljava -war f -o s.war m.cmj ...`
where `f` is the file to be used as the webapp descriptor

`ocamlbuild` (extended)

recognizes the `ocamljava`-specific extensions and tags for the additional command-line switches, plus:

`use_javalib`	for the Java library
`use_concurrent`	for the concurrent library

Post-compilation optimization

A compiled `jar` file can be optimized through
```
ocamljar [options] in.jar out.jar
```
possible options include:

`-no-backtrace v`	to set backtrace support
`-no-debug v`	to set debug support
`-no-dynlink v`	to set dynlink support
`-no-runtime-lock v`	to set runtime lock use
`-no-signals v`	to set signals support
`-no-unused-globals v`	to set removal of unused globals
`-unsafe v`	to set use of *unsafe* data containers
`-war`	if passed file is a `war` archive

where `v` can be either `false` or `true`

Wrappers generation

Wrappers for elements of a module can be generated by:
```
ocamljava -c m.mli
ocamljava -c m.ml
ocamljava -o p.jar m.cmj
ocamlwrap m.cmi
```
resulting in a file named `MWrapper.java` allowing to access the OCaml elements

Typer extension

Mapping of types

Java type	OCaml type	note
`boolean`	`java_boolean = bool`	
`byte`	`java_byte = int`	
`char`	`java_char = int`	
`double`	`java_double = float`	
`float`	`java_float = float`	
`int`	`java_int = int32`	
`long`	`java_long = int64`	
`short`	`java_short = int`	
`pack.Class`	`pack'Class java_instance`	(1)
	`pack'Class java_extends`	(2)

(1) used to designate exactly an instance of `pack.Class`
(2) used to designate an instance of `pack.Class` or any subtype

Instance creation

```
let obj = Java.make "pack.Class(sign)" params
```

Method calls

```
Java.call "pack.Class.meth(sign)" inst params
Java.call "pack.Class.stat(sign)" params
```
`Java.exec` is similar to `Java.call` but ignores result
`Java.chain` is similar to `Java.call` but returns instance

Field accesses

```
let val = Java.get "pack.Class.field:type" inst
Java.set "pack.Class.field:type" inst val
let val = Java.get "pack.Class.stat:type" ()
Java.set "pack.Class.stat:type" val
```

Type checks

```
let cls = Java.get_class inst
let bool_val = Java.instanceof "pack.Class" inst
let inst' = Java.cast "pack.Class" inst
```

Sugar

Any type in a signature can be replaced with an underscore ("_") as long as there is no ambiguity; a dash ("-") can be used instead of a whole signature as long as there is no ambiguity

`open Package'pack` is equivalent to `import pack.*;`, and `open Class'pack'Class` is equivalent to `import pack.Class;`, both allowing to use simple class names instead of fully-qualified class names

Proxies

```
Java.proxy "pack.Interface" (object
  method m1 ... = ...
  method m2 ... = ...
end)
```

builds an instance implementing the interface declared as:

```
package pack;
public interface Interface {
  ... m1(...);
  ... m2(...);
}
```

Exceptions

```
exception Java_exception of j'l'Exception java_instance
exception Java_exception of j'l'Error java_instance
```
are used to respectively represent Java exceptions and errors; both can be caught as regular OCaml exceptions

`Java.throw inst` is used to raise a Java exception; `inst` must be an instance of `java.lang.Throwable`

Main modules of `javalib.cmja`

`Java`	basic functions
`JavaObject`	bindings for `java.lang.Object`
`JavaXyzArray`	arrays of `Xyz` values (one for each primitive type plus one for references)
`JavaArray`	generic representation of arrays
`JavaIOStreams`	conversion between Java streams and OCaml channels
`JavaApplet`	type definitions for the various applet kinds
`JavaServlet`	type definitions for the various servlet kinds