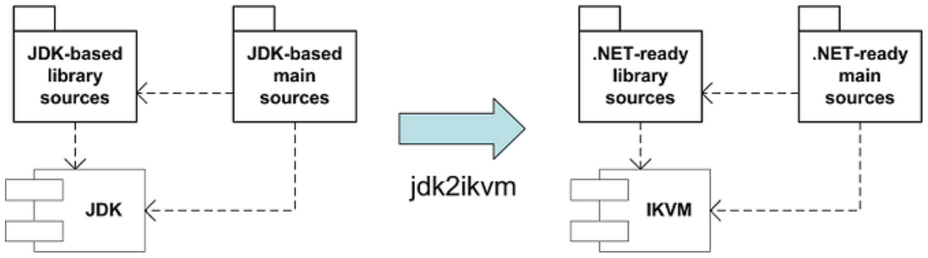
A preview of Scala.NET: Cross-platform development the Scala way

© Miguel Garcia, LAMP, EPFL
http://lamp.epfl.ch/~magarcia

May 16th, 2011

Abstract

Although Scala.NET hasn't achieved yet feature parity with the flagship Scala compiler, it nonetheless already knows one neat trick: how to target JDK and .NET from a single codebase. The open-source tools used for this task on the Scala side are pre-release yet mature enough that the daily build of Scala.NET results from the same sources used to build Scala on JVM. Given that Scala.NET is a .NET-only compiler (no cross-compilation, just linking and emitting assemblies) there was a need to *convert* the JDK-based sources of the original compiler to use .NET APIs. That conversion applies the same recipe as the IKVM compiler (`ikvmc`), with the difference that `ikvmc` is a bytecode-level tool while `jdk2ikvm` is source-level. In these notes, we show (a) how to develop Scala.NET apps from scratch; as well as (b) how to re-target the sources of Scala applications from JDK to .NET, with near-zero manual steps.



Contents

1 Cross-platform development the Scala way

Platform migration has never been easy but the rewards are apparent, especially for *automated, continuous migration from a single codebase* (leaving out just one of these features makes costs skyrocket, due to the delays and error-proneness associated with manual steps).

For all their benefits, software platforms like JDK and the .NET Framework don't contribute by themselves to making migration any easier (they have increased the reward however) because of two factors: (a) lack of a unified programming language across platforms; and (b) large *contact surface* in terms of APIs offered by JDK and .NET. Regarding the first item, there wasn't much developers could do (short of writing C programs). Addressing the second item (for example, by encapsulating platform dependencies in well-defined components) has proved elusive in practice: depedencies have the tendency to creep in (including those of the `com.sun.*` variety).

Our approach to migration addresses both concerns above: Scala compilers are in the unique position of supporting the exact same language on both plaftorms. If the language is the same then porting an existing application is just a matter of replacing JDK calls with .NET ones, right? Yes, exactly, not as a one-time effort performed by someone well-versed in the intricacies of both platforms, but *automatically, upon changesets on the single codebase.*

That brings us to the IKVM library. It's an implementation of JDK for .NET, excelling at two seemingly competing goals: first and foremost, faithfully preserving JDK semantics, all while shunning software emulation in favor of CLR capabilities for better performance. As an example of the latter, Java strings are not emulated but native `System.String` instances are used. Same goes for exceptions and arrays. This desirable feature has a ripple effect throughout the API exposed by IKVM, that has to be taken into account when targeting it (don't worry: `jdk2ikvm` does it for you). For example, a `System.String` can't possibly implement interface `java.lang.CharSequence`, unlike Java strings.

Although we focus on one particular approach to cross-platform development with Scala, we mention that in green-field scenarios another strategy might be preferable. There, the cost of encapsulating platform dependencies behind common APIs can be spread over the project lifetime (for example, by standardizing from the start on `scala.io` as a platform-neutral facade to filesystem functionality). While `jdk2ikvm` does not help with that initial porting (in the example, from `java.io` to `scala.io`), it does not stand in the way either, and thus an incremental migration strategy away from JDK APIs in favor of the Scala SDK is also possible.

1.1 Download

Precompiled versions of the Scala library and the compiler for .NET can be obtained via SVN, in the `bin` folder of the preview:

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala-experimental/trunk/bootstrap
```

2 Developing Scala.NET apps from scratch

Scala.NET does not as of this writing (`2011-05-16`) fully support the all-important Generics that pretty much every modern assembly relies on. What *can* be compiled against are: `scalalib.dll`, IKVM, and libraries containing backwards compatible versions of generified APIs. By the way, IKVM does not use generics because at bytecode level there are no generics in OpenJDK.

(Please be assured that if you see us writing caveats like the one above, it's because we're already hard at work to add Generics to Scala.NET **ASAP** :-)

2.1 An interim solution for IDE support

Visual Studio supports step-debugging of Scala.NET programs (Sec. 4) but not their editing. Until a VS extension for Scala arrives, Scala.NET programs can be developed with an existing (JVM-based) IDE:

1. programs using just the Scala library compile unmodified on Scala.NET, and thus a Scala IDE can be used to develop them.

2. programs using (non-generic) APIs from .NET assemblies can also be written with the help of a JVM-based Scala IDE, provided that a *stub* is provided for that API. Figure 1 depicts the development experience. As long as the IDE can be made aware about the target .NET APIs, the illusion will work. Two "*stubbing*" alternatives exist: `md2src` and `ikvmstub`.

The `md2src` alternative is described in some depth in its write-up[1]:

> *It would be useful if .NET types could be represented in a format consumable by current Scala IDEs. That way, a (JVM-based) IDE can be used to develop Scala.Net apps (all navigation and editing goodies included) offloading compilation to the Scala.Net compiler. These notes showcase `md2src`, a tool that allows doing just that. For each `.dll` library that an app will use, `mdsrc` outputs `.scala` files with type stubs that keep the IDE happy. The Scala.NET compiler instead finds out about external types in `.dlls` themselves.*

A precompiled version of `md2src` is available for download[2]. At the command-line, just provide the file-path to the assembly to "*stub-ify*", making sure that `mscorlib.dll` can be found in the same folder as `<assembly-file>`:

```
scala -cp %MD2SRC_JAR% scala.tools.msil.md2src.Main <assembly-file>
```

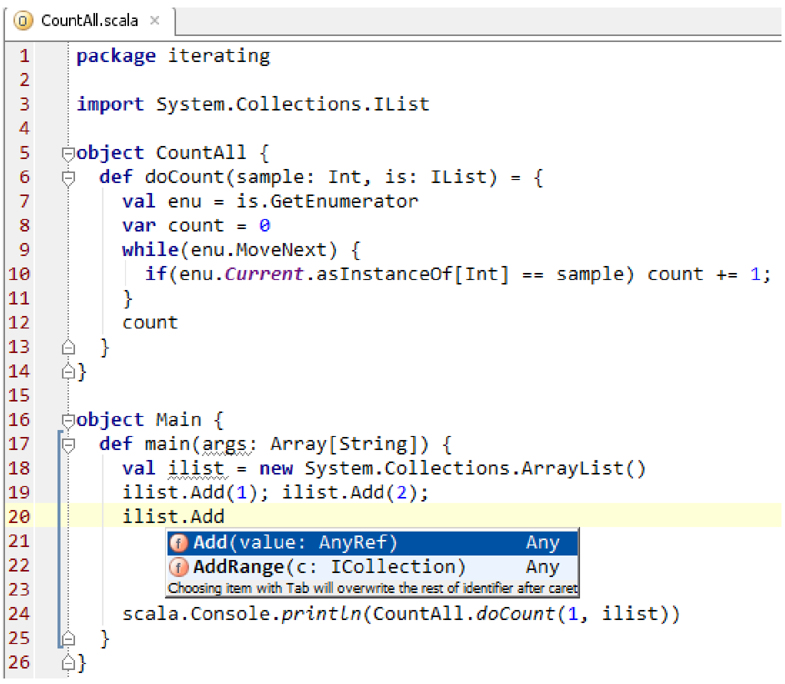
3 Porting apps originally developed for JDK

3.1 Basic usage

A precompiled version of `jdk2ikvm` can be downloaded from `http://lamp.epfl.ch/~magarcia/jdk2ikvm/soft/jdk2ikvm.jar`

Using `jdk2ikvm` to convert a tree of sources feels much like compiling them on JDK. The template of a typical invocation looks as follows:

[1] `http://lamp.epfl.ch/~magarcia/ScalaNET/2011Q2/TestDriveMD2SRC.pdf`
[2] `http://lamp.epfl.ch/~magarcia/jdk2ikvm/soft/md2src.jar`



CountAll.scala

```
package iterating

import System.Collections.IList

object CountAll {
  def doCount(sample: Int, is: IList) = {
    val enu = is.GetEnumerator
    var count = 0
    while(enu.MoveNext) {
      if(enu.Current.asInstanceOf[Int] == sample) count += 1;
    }
    count
  }
}

object Main {
  def main(args: Array[String]) {
    val ilist = new System.Collections.ArrayList()
    ilist.Add(1); ilist.Add(2);
    ilist.Add
          Add(value: AnyRef)          Any
          AddRange(c: ICollection)    Any
          Choosing item with Tab will overwrite the rest of identifier after caret

    scala.Console.println(CountAll.doCount(1, ilist))
  }
}
```

Figure 1: Piggybacking on JVM-based IDEs to develop Scala.NET apps, Sec. 2.1

```
// all in one line, line-breaks inserted for readability only

scalac -Xplugin c:\myplugins\jdk2ikvm.jar
      -P:jdk2ikvm:output-directory:%OUT_FOLDER% -d %OUT_FOLDER%
      -Ystop-after:superaccessors -Yrangepos
      . . . sources
```

Additional compiler options are there to let `scalac` know where `jdk2ikvm` is located and the folder that will hold converted files. Two more options (`-Ystop-after` and `-Yrangepos`) reveal the way `jdk2ikvm` is implemented: it's a *compiler plugin*[3], i.e. it performs additional processing between certain compilation phases (in our case, it serializes to disk adapted sources). After that, `scalac` stops.

The real compilation is done with `scalacompiler.exe`:

```
scalacompiler.exe -Ydebug -d %MSIL_FOLDER%
  -target:msil
  -Ystruct-dispatch:no-cache -no-specialization
  -Xassem-name %OUT_ASSEMBLY%
  -Xassem-extdirs %DLL_FOLDER%
  -Xshow-class %MAIN_CLASS%
  . . . (converted) sources
```

Some options are specific to Scala.NET, and `scalacompiler.exe -X` reveals their purpose:

```
. . .
  -Xassem-extdirs <dirs> (Requires -target:msil) List of directories containing assemblies.
```

[3]http://www.scala-lang.org/node/140

```
  -Xassem-name <file>   (Requires -target:msil) Name of the output assembly.
  -Xassem-path <path>   (Requires -target:msil) List of assemblies referenced by the program.
. . .
```

The last option shown in the template above (`-Xshow-class`) is a bit idiosyncratic, in that it has been borrowed from `forJVM` compilation mode and retooled for another purpose: in Scala.NET it gives the fully-qualified name of the class with the entry `main` method, thus indicating that an `.exe` should be emitted. Leaving it out results in a `.dll` being generated.

Finally, `-Ydebug` is there just to show compilation progress, and `-d` is naturally the output folder, with a twist: the output of Scala.NET is a text file with IL instructions, of the kind that the `ilasm` assembler knows how to deal with (yes, that's an additional compilation step, but usually way faster than one might assume). A future version of Scala.NET will directly emit binary assemblies. In the meantime ... how to run `ilasm` is the topic of `http://msdn.microsoft.com/de-de/library/496e4ekx.aspx`

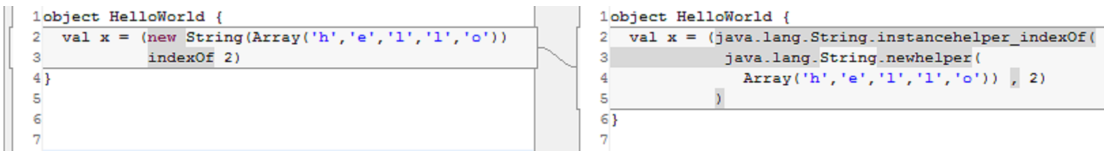
The simplest way to get started involves having in `-Xassem-extdirs` the same assemblies that `scalacompiler.exe` requires to run (`scalalib.dll`, the IKVM library, and a few others). This way, you may be getting assemblies not really needed, but on the other hand one makes sure that none of the assemblies that pretty much every Scala.NET program needs is missing. Additionally, after assembling a Scala.NET program, it makes sense to run `peverify` (bytecode verification) right there, before the program's first run.

As already mentioned, precompiled versions of the Scala library and the compiler for .NET can be obtained via SVN, in the `bin` folder of the preview:

```
svn co http://lampsvn.epfl.ch/svn-repos/scala/scala-experimental/trunk/bootstrap
```

3.2 What `jdk2ikvm` does behind the scenes

The bread and butter of `jdk2ikvm` transformations are callsite rewritings:



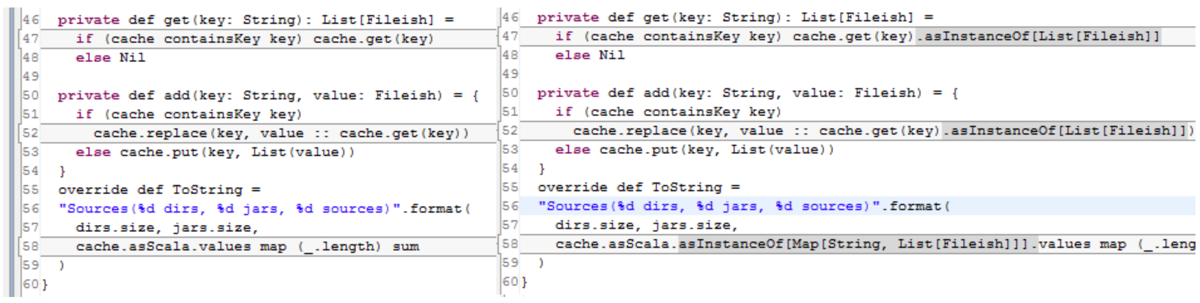
```
object HelloWorld {
  val x = (new String(Array('h','e','l','l','o'))
             indexOf 2)
}
```

```
object HelloWorld {
  val x = (java.lang.String.instancehelper_indexOf(
             java.lang.String.newhelper(
               Array('h','e','l','l','o')) , 2)
           )
}
```

`jdk2ikvm` is only 2KLOC heavy, and thus there are a few uncommon cases where it leaves the input as-is, and manual rewriting is needed afterwards. At the latest, this is discovered when Scala.NET refuses to compile, but more often `jdk2ikvm` will have emitted a warning beforehand:

```
Z:\scalaproj\src\compiler\scala\tools\ant\sabbus\CompilationFailure.scala:12:
warning: [jdk2ikvm] couldn't substitute type at scala.`package`.Exception with java.lang.Exception
case class CompilationFailure(message: String, cause: Exception) extends Exception(message, cause)
                                                                  ^
```

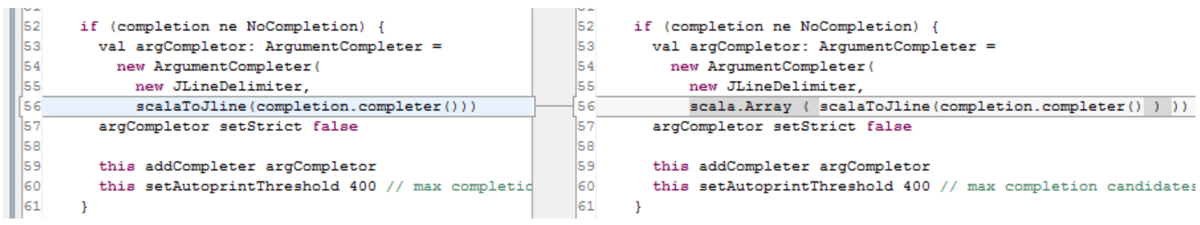
Sometimes, the consequences of IKVM's erased types exceed what `jdk2ikvm` automates (the manually rewritten version is shown on the left):



```
private def get(key: String): List[Fileish] =
  if (cache containsKey key) cache.get(key)
  else Nil

private def add(key: String, value: Fileish) = {
  if (cache containsKey key)
    cache.replace(key, value :: cache.get(key))
  else cache.put(key, List(value))
}
override def ToString =
"Sources(%d dirs, %d jars, %d sources)".format(
  dirs.size, jars.size,
  cache.asScala.values map (_.length) sum
)
}
```

```
private def get(key: String): List[Fileish] =
  if (cache containsKey key) cache.get(key).asInstanceOf[List[Fileish]]
  else Nil

private def add(key: String, value: Fileish) = {
  if (cache containsKey key)
    cache.replace(key, value :: cache.get(key).asInstanceOf[List[Fileish]])
  else cache.put(key, List(value))
}
override def ToString =
"Sources(%d dirs, %d jars, %d sources)".format(
  dirs.size, jars.size,
  cache.asScala.asInstanceOf[Map[String, List[Fileish]]].values map (_.leng
)
}
```

Shown below, an interplay between (a) IKVM giving array type for a repeated param in a JDK signature; and (b) a single argument, which was handled as-is by the Scala compiler on JDK, but now has to be explicitly packed in an array (usually, `jdk2ikvm` performs this rewriting, but not in this case):

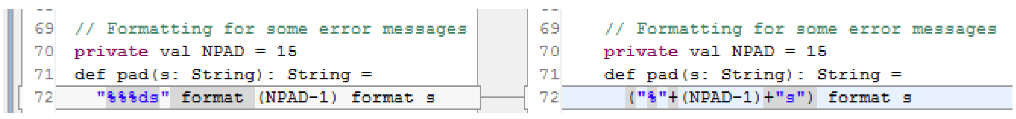


```
    if (completion ne NoCompletion) {
      val argCompletor: ArgumentCompleter =
        new ArgumentCompleter(
          new JLineDelimiter,
          scalaToJline(completion.completer()))
      argCompletor setStrict false

      this addCompleter argCompletor
      this setAutoprintThreshold 400 // max completic
    }
```

```
    if (completion ne NoCompletion) {
      val argCompletor: ArgumentCompleter =
        new ArgumentCompleter(
          new JLineDelimiter,
          scala.Array ( scalaToJline(completion.completer() ) ))
      argCompletor setStrict false

      this addCompleter argCompletor
      this setAutoprintThreshold 400 // max completion candidate:
    }
```

Based on our experience porting the Scala compiler, manual retouching was needed for less than 1% of all sources, and then only in a few places (we have several examples to report only after porting a couple thousand files).

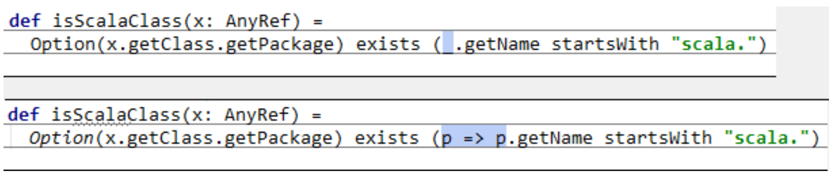
Rather than extending `jdk2ikvm` (Sec. 5) an easier strategy is "workarounding". For example, both versions below behave identically on JDK, but for some reason a literal formulation of `"%%%ds" format (NPAD-1) format s` just won't work on IKVM. Solution: simplify the code at the source:



```
  // Formatting for some error messages
  private val NPAD = 15
  def pad(s: String): String =
    "%%%ds" format (NPAD-1) format s
```

```
  // Formatting for some error messages
  private val NPAD = 15
  def pad(s: String): String =
    ("%"+(NPAD-1)+"s") format s
```

As a last example, the closure's argument can't be a wildcard when some `java.lang.String` method is called:



```
def isScalaClass(x: AnyRef) =
  Option(x.getClass.getPackage) exists (_.getName startsWith "scala.")
```

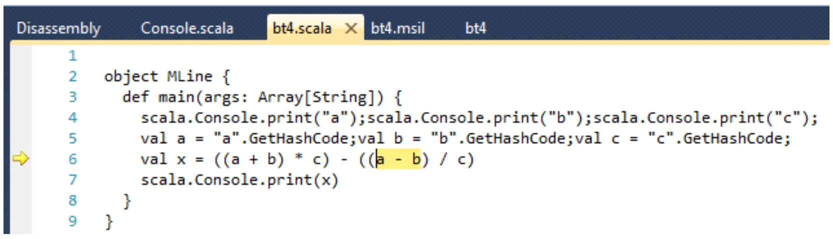
```
def isScalaClass(x: AnyRef) =
  Option(x.getClass.getPackage) exists (p => p.getName startsWith "scala.")
```

because otherwise a wildcard will show up inside an actual argument (where `p.getName` appears below), and that's a no-no in Scala:

```
// code emitted by jdk2ikvm
def isScalaClass(x: AnyRef) =
  Option(_root_.java.lang.Object.instancehelper_getClass(x).getPackage) exists
  (p => _root_.java.lang.String.instancehelper_startsWith(p.getName, "scala."))
```

Summing up: not many cases to manually retouch, most of them can be patched once and for all in the files given as input to `jdk2ikvm`, and those uncommon remaining cases will be clearly pointed out by Scala.NET.

4 Debugging in Visual Studio



```

object MLine {
  def main(args: Array[String]) {
    scala.Console.print("a");scala.Console.print("b");scala.Console.print("c");
    val a = "a".GetHashCode;val b = "b".GetHashCode;val c = "c".GetHashCode;
    val x = ((a + b) * c) - ((a - b) / c)
    scala.Console.print(x)
  }
}
```

In order to step-debug over Scala sources:

1. "*File / New Project*"
2. "*Other Project Types / Visual Studio solutions / Blank solution*"
3. right click on the new solution, "*Add existing project*" and pick `myprogram.exe` (on the same folder, `myprogram.pdb` should be found)
4. From the main menu, "*Debug / Options and Settings*", mark "*Enable address-level debugging*"
5. ready to go, "*Step into new instance*". Besides line breakpoints, exception breakpoints are very useful ("*Debug / Exceptions*").

Another write-up[4] gives details on debugging at the `.msil` (bytecode) level.

5 Further resources

Some venues for further exploration:

- Typically, the larger the application, the more varied its use of JDK APIs. Although IKVM is the most complete implementation of JDK on .NET, it is still the case that some APIs are not supported. Details at `http://www.ikvm.net/`
- The conversion recipe of `jdk2ikvm` is covered in a 30-page write-up[5].
- The website `http://lamp.epfl.ch/~magarcia/jdk2ikvm/` points to the Subversion repository for `jdk2ikvm`, and discusses how to extend or adapt the tool to other migration scenarios (say, from `java.io` to `scala.io`).

Acknowledgement

Big thanks to Jeroen Frijters for creating IKVM, `http://www.ikvm.net/`.

[4]`http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/ikvmify4.pdf`
[5]`http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/ikvmify2.pdf`