Mental adjustments demanded by IKVM's *Object Model Mapping*

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

October 14th, 2010

Abstract

These notes are the first in a three-part series describing the results of automating the "JDK to IKVM" conversion of Scala sources (in particular those of the compiler, to bootstrap Scala.NET). In this part, we automate the recipe by patching a prototype "Scala.IKVM" compiler (in retrospect, it would have saved time to start developing the `jdk2ikvm` compiler plugin all right). On the plus side, the prototype allows validating the recipe. Although compilation succeeds and .NET binaries are emitted for both the Scala library and the compiler, there remain bytecode verification errors, including one that prevents successful runs of `scala-compiler.exe`: the JDK-based sources contain usages of `classOf[tpe]` for which the .NET version should return a `System.Type` or an (IKVM's) `java.lang.Class` instance, depending on whether (a) "platform reflection" is intended; or (b) "IKVM reflection" is intended. As an example of the former, `CleanUp` should build trees to cache `System.Type` instances. As an example of the latter, IKVM's `Object.getClass` returns a `j.l.Class`. Part 3 of this series will cover sorting out those usages, and Part 2 covers bugs fixed along the way (Scala.NET bugs which are independent of the JDK to IKVM conversion).

Contents

1 Remapped types **3**
1.1 `java.lang.Object` according to IKVM 4
1.1.1 `clone()` overrides and (super) invocations 5
1.1.2 Adding the `java.lang.Cloneable` interface 6
1.1.3 `clone()` invocations on arrays 6
1.1.4 Translation of `finalize()` 7
1.1.5 FYI: Callsites to detour 7
1.2 `java.lang.String` 7
1.3 Type casts and checks 10

2 Fulfilling the `j.l.Object` contract without inheriting from `j.l.Object` **12**
2.1 The problem: Illegal inheritance (two classes to extend from) 12
2.1.1 Let `ScalaObject` perform the detouring (doesn't compile) . . . 13
2.1.2 Let an implicit + view detour (doesn't cope with overrides) . . 13
2.1.3 Let `ScalaObject` check fulfillment of the `j.l.Object` contract . 13

2.2 Part of the solution: entering method symbols for the `j.l.Object` contract, replacing invocations afterwards . . . 14
2.2.1 `Definitions.scala` part of the story . . . 15
2.2.2 `GenMSIL` part of the story . . . 16
2.3 Method co-overrides for the `Object` contract . . . 16

3 Adding missing overrides that JDK's `Object` used to provide 17
3.1 Backround . . . 17
3.2 Mechanics: `SyntheticMethods` now adds a few `DefDef`s more . . . 18
3.3 What-if missing: diagnosing with `peverify`'s help . . . 18

4 Adding missing overrides for "*implied interfaces*" 18
4.1 Adding missing `IEnumerable` overrides . . . 18
4.2 Adding missing `IDisposable` overrides . . . 19

5 Parser-level rewritings 19
5.1 Rewriting "`System.`" selectors . . . 20
5.2 Rewriting `new String` . . . 21
5.3 Erasing type arguments to IKVM classes . . . 22
5.3.1 Non-existentials . . . 22
5.3.2 Existentials . . . 23
5.4 Ignoring `@throws` and `@serializable` . . . 24

6 Manual rewritings 25
6.1 `length` vs. `length()` . . . 25
6.2 `Array.empty[T]` . . . 25
6.3 `ArrayRuntime` should be public . . . 25

7 Preprocessor: Magic around interfaces 26
7.1 Implied interfaces . . . 26
7.1.1 `ikvmc` and the `IEnumerable` implied interface . . . 26
7.1.2 `ikvmc` and the `IDisposable` implied interface . . . 27
7.2 Extra interface (`j.l.Comparable`) . . . 28
7.3 Ghost interfaces . . . 28

8 Preprocessor: Exception handling 29
8.1 Case (1) Originally `catch Throwable` . . . 30
8.2 Case (2) Originally `catch Exception` or `catch Error` . . . 31
8.3 Case (3) Otherwise . . . 31

9 Preprocessor: Serialization 32
9.1 Case (A): `java.io.Externalizable` . . . 32
9.2 Case (B): Base class is serializable . . . 32
9.3 Case (C): Otherwise . . . 32

1 Remapped types

`j.l.String` is sealed in both its JDK and IKVM versions (in the latter, it's also abstract). Consequences:

- when the receiver is known to be `String` the invoked method can't possibly have been overriden. In contrast, IKVM's `j.l.Object` has both final methods (`getClass()`, `notify()`, `notifyAll()`, `wait(...)`) and non-final ones (`hashCode()`, `equals(Object)`, `toString()`, and `finalize()`) in keeping with the JDK original.
- After translation, `j.l.String` can't appear anywhere as declared type, nor there will be any instances that conform to `j.l.String` at runtime. In contrast, for `ikvmc`-emitted assemblies, there could be `j.l.Object`-conforming instances.
  - However, after translation, `new Object` instantiates a `System.Object`, and `ScalaObject` extends `System.Object`.
  - This explains why `ikvmc` uniformly rewrites `j.l.String` occurrences as `System.String`, while `j.l.Object` standalone occurrences are left as-is (except in formal params which are upcast to `System.Object`).

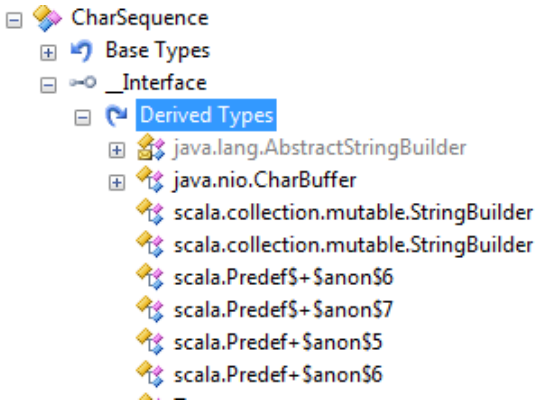
Believe it or not, there are receivers which may receive `j.l.Object` method invocations (before translation) but do not inherit from `j.l.Object` (after translation). In other words, after translation, their actual type does not conform to IKVM's `j.l.Object`. Examples:

- arrays
- strings
- an instance with `System.Object` as actual type, as actual argument to a formal param whose before-translation declared type is `j.l.Object` (an argument usually provided by non-IKVM-aware code, "interoperability")
- instances of `j.l.Throwable` (in IKVM, `j.l.Throwable` derives from `System.Exception` and *not* from `j.l.Object`)
- in general, any `System.Exception`-conforming instance does not derive from `j.l.Object` (and thus does not conform to it).

A *ghost interface* is a JDK interface for which no same-name *interface* but a *struct* exists in IKVM. The term refers to: `java.lang.CharSequence`, `java.lang.Cloneable`, and `java.io.Serializable`. Details in Sec. 7.3. IKVM supports serialization as described in Sec. 9.

Summary of *extra interfaces* and *ghost types*:

- IKVM types with a nested `__Helper` class: `java.lang.Comparable`
- IKVM types with a nested `__Interface` interface: `java.io.Serializable`, `java.lang.Cloneable`, `java.lang.CharSequence`

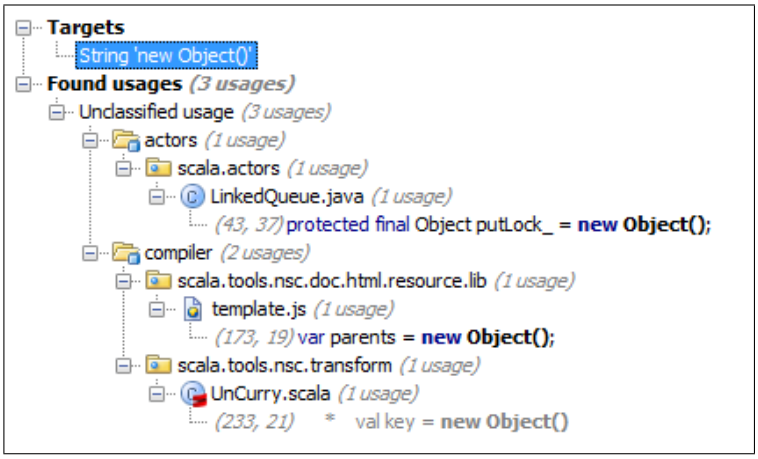


1.1 `java.lang.Object` according to IKVM

1. `clone()` is covered in Secs. 1.1.1 to 1.1.3

2. `finalize()` is covered in Sec. 1.1.4

3. Both the JDK and the IKVM versions of `j.l.Object` have a single (the default) constructor. At the bytecode level, `ikvmc` invokes it. However the favored declared type after translation is the more "interoperable" `System.Object` (instances of subclasses of `java.lang.Object` do get as declared type the instantiated type, rather than being "upcast" by `ikvmc` to `System.Object`). At the source-level, "`new Object`" instantiates whatever the type alias in `Predef` has as RHS (thus we wouldn't want `System.Object` as RHS, but doing otherwise leads to a dead-end, Sec. 2.1). There's only one place (in `actors`) where "`new Object`" occurs:



4. *Instance method invocations* (targeting a `j.l.Object` method through a `System.Object` reference) are detoured to `instancehelpers` (they take a `System.Object` as first argument).

   - That is, all instance method invocations other than `clone()`. And we know it's not possible to direcly invoke `finalize()`.

Figure 1: Places where `java.lang.Object` shows up in the sources

- `ikvmc` leaves as-is super-invocations on the `j.l.Object` contract (e.g., "`super.hashCode()`" is translated by `ikvmc` as "`base.hashCode()`"). Those callsites are not detoured to `instancehelper`s. We won't detour them either, because for each IClass defining, say, `hashCode` we'll emit a `GetHashCode` "co-override" which delegates to the developer-provided implementation. Same thing with `toString` and `finalize` (they get `ToString` and `Finalize` resp.)
- There are `instancehelper`s for `j.l.Object`'s `final` methods (`getClass()`, `notify()`, `notifyAll()`.

5. *Standalone occurrences.* In formal params, `ikvmc` "upcasts" the declared type from `j.l.Object` to `System.Object` (for interoperability). Other than that, `ikvmc` leaves as-is specific occurrences of `java.lang.Object` (for example, in "`extends java.lang.Object`" and "`new java.lang.Object()`").

   - In order to stress this point: except for occurrences in formal params, *no* standalone occurrences of `j.l.Object` (shown in Figure 1) is rewritten by `ikvmc` to `System.Object`.
   - BTW, usages of the form `classOf[java.lang.Object]` do not require generics.

1.1.1 `clone()` overrides and (super) invocations

In a nutshell, no special-case rewriting is needed, neither for `clone()` overrides nor for invocations, be those invocations on `super` or not. However other rewritings are necessary as covered in Sec. 1.1.2 and Sec. 1.1.3.

Regarding `clone()` and `finalize()`, IKVM's `j.l.Object` declares both:

- "`protected internal virtual`" versions of them, corresponding to the original `protected` counterparts
- "`internal static`" versions of the form `instancehelper_clone` and `instancehelper_finalize`. I don't know who invokes them (we won't).

`ikvmc` performs no special-case transforms for `clone()` overrides or invocations:

```
private void testClone() throws CloneNotSupportedException { new Test().clone(); }
@Override
protected Object clone() throws CloneNotSupportedException { return super.clone(); }
```

gets translated by `ikvmc` as:

```
private void testClone() { new Test().clone(); }
protected internal override object clone() { return base.clone(); }
```

FYI: `MemberwiseClone()` returns a shallow copy (i.e., object references point to the same object as in the original) and is invoked by IKVM's `clone()` version in `j.l.Object`, which reads:

```
.method famorassem hidebysig newslot virtual instance object clone() cil managed
{
  .maxstack 8
  L_0000: ldarg.0
  L_0001: dup
  L_0002: call bool java.lang.Cloneable::IsInstance(object)
  L_0007: brtrue.s L_000b
  L_0009: pop
  L_000a: ldnull
  L_000b: brtrue L_0016
  L_0010: newobj instance void java.lang.CloneNotSupportedException::.ctor()
  L_0015: throw
  L_0016: ldarg.0
  L_0017: call instance object [mscorlib]System.Object::MemberwiseClone()
  L_001c: ret
}
```

1.1.2 Adding the `java.lang.Cloneable` interface

Doing so does not affect how the `clone()` override is translated, but results in adding a CLR implicit conversion. We use a view instead, the full recipe appears under "Magic for interfaces". Additionally, `java.lang.Cloneable.__Interface` is added as super interface while `j.l.Cloneable` is deleted.

1.1.3 `clone()` invocations on arrays

"`arr.clone()`" is not translated as for an `Object` receiver. Instead, the instance method `System.Array.Clone()` should be invoked, as the following `ikvmc` input-output shows:

```
private void testCloneArrays() throws CloneNotSupportedException {
  String[] letters = new String[] { "a", "b", "c"};
  letters.clone();
}
```

gets translated as:

```
private void testCloneArrays()
{
  string[] textArray1 = new string[] { "a", "b", "c" };
  textArray1.Clone(); /*- this is System.Array.Clone() */
}
```

1.1.4 Translation of `finalize()`

In addition to the no-frills translation for the method itself, a detouring co-override (`Finalize()`) is added.The special-casing for `finalize` overrides can be seen in an example. Given the following in class `Test`:

```
@Override
protected void finalize() throws Throwable {
  super.finalize();
}
```

the co-override `Finalize()` should also be emitted:

```
protected internal override void finalize()
{
  base.finalize();
}

[HideFromJava]
protected override void Finalize()
{
  if (!IKVM.Runtime.ByteCodeHelper.SkipFinalizer())
  {
    try { this.finalize(); } // callvirt instance void Test::finalize()
    catch { }
  }
}
```

1.1.5 FYI: Callsites to detour

Places where invocations to selected `Object` instance methods show up in the sources:

- `getClass()`, Figure 2
- `clone()`, Figure 3
- `notify()`, `notifyAll()`, and all of the `wait()` overloads are invoked in `actors` only.

1.2 `java.lang.String`

1. All standalone usages of `j.l.String` (there are a few, Figure 4) can be safely reformulated as `scala.String`. That type alias in `Predef` should denote `System.String`. In general, any "`java.lang.String`" references should be rewritten to `System.String`
2. *Accesses to static members* are left as-is (both method invocations such as `format(...)` and read accesses to the field `CASE_INSENSITIVE_ORDER`). For readability, the `java.lang.String` FQN should appear as prefix to a static access. However, those statics can be added to `JLStringDetour` for convenience (Listing 2 on p. 34).
3. *Constructor invocations* are detoured to the `newhelper` with the same argument-signature (in all of IKVM, the only class with `newhelper(...)` methods is `j.l.String`). Example for `new String()`:

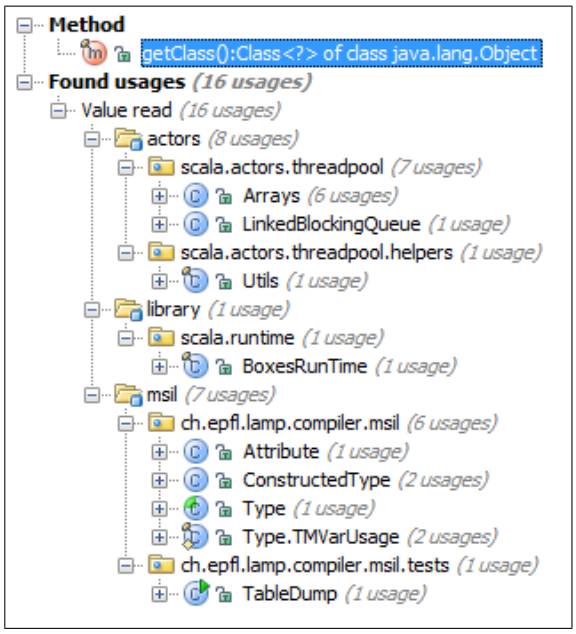


Figure 2: Places where `getClass()` is invoked, Sec. 1.1.5

```
// IKVM library
public static string newhelper()
{
  return string.Copy("");
}
```

4. *Instance method invocations* are detoured to `instancehelper`s (they take a `System.String` as first argument).

   (a) There are `j.l.String`-specific `instancehelper`s for those instance methods inherited from `j.l.Object` (`hashCode()`, `equals(Object)`, and `toString()`, but none for `finalize()`). Whenever `ikvmc` can determine statically that the receiver is a string, detouring lands there.

   (b) For example, `ikvmc` translates `"zzz".equals("123")` into `java.lang.String.instancehelper_equals("zzz", "123")`. The following metadata (`NameSig`) instructs `ikvmc` to perform this rewriting:

```
[Modifiers(Modifiers.Public),
 NameSig("equals", "(Ljava.lang.Object;)Z")] /*- this makes ikvmc detour invocations into here */
public static bool instancehelper_equals(string @this, object anObject)
{
  string text1 = @this.ToString;
  return @this.Equals(anObject); // callvirt instance bool [mscorlib]System.String::Equals(object)
}
```

   (c) However, even without that knowledge about the receiver's type, a correct translation is possible. As can be seen from inspecting the `instancehelper`s in `j.l.Object`, when their first argument is a string,

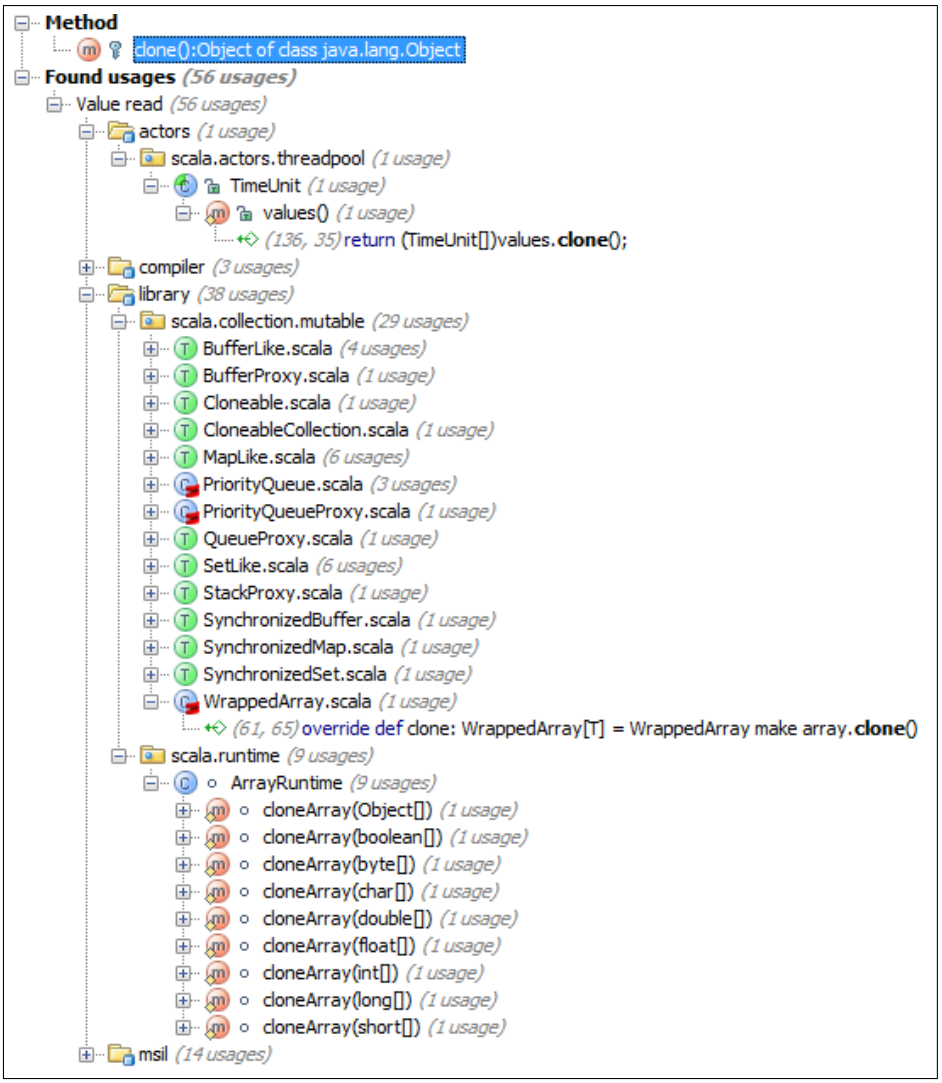


Figure 3: Places where `clone()` is invoked, Sec. 1.1.5

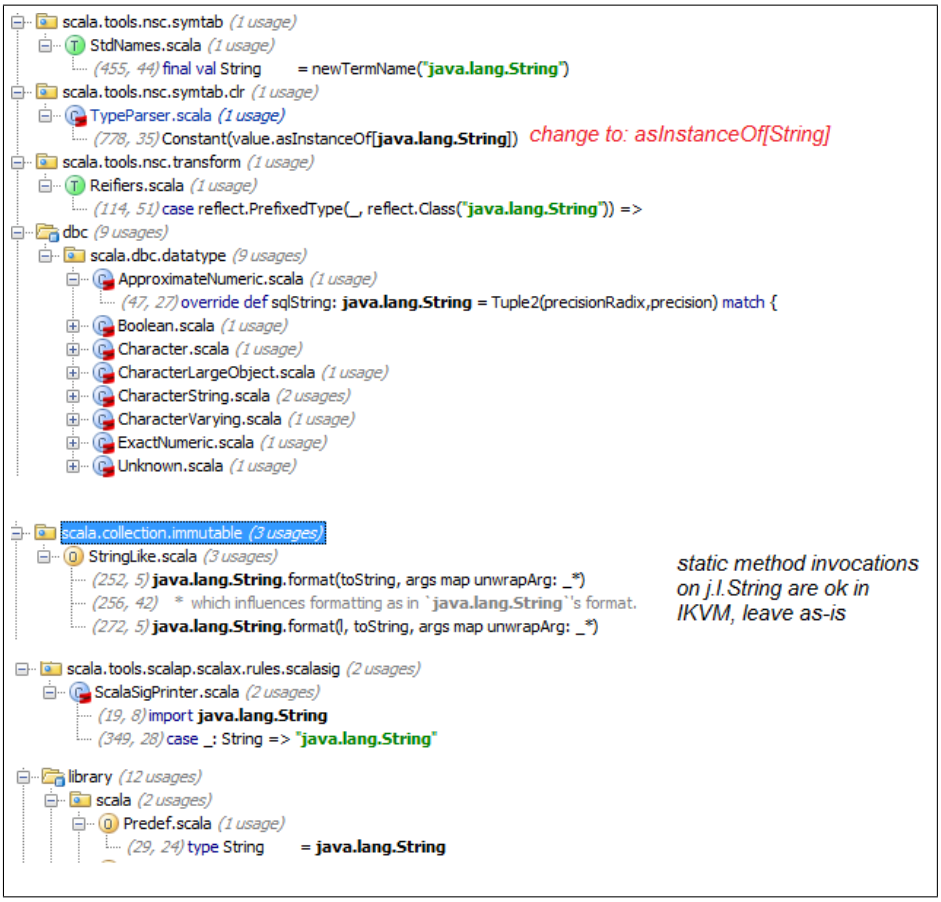


Figure 4: Places where `java.lang.String` shows up in the sources

the same method bodies as for `j.l.String`'s `instancehelpers` will run. It just costs an additional type-check.

Given that after applying the first three transformations the receiver is a `System.String`, it is possible to have a `Predef` view into `JLStringDetour` (Listing 2 on p. 34) whose instance methods detour to `instancehelpers`, thus slimming down the preprocessor.

1.3 Type casts and checks

We're talking about: `isInstanceOf[C]`, `asInstanceOf[C]`, `classOf[C]`

There are four cases to consider when translating "`arg.isInstanceOf[C]`" and "`arg.asInstanceOf[C]`":

1. `C` is ghost:
   `arg.isInstanceOf[java.lang.CharSequence]`
   → `java.lang.CharSequence.IsInstanceOf(arg)`

   `arg.isInstanceOf[java.lang.Cloneable]`

Listing 1: Slimming down the transformations for serialization, Sec. 9.2 and Sec. 9.3

```
// Scala.NET code to be compiled against IKVM

implicit def refToStructSerializable
(i: java.io.Serializable.__Interface): java.io.Serializable = {
    val c : java.io.Serializable = new java.io.Serializable() // default init
    c.__<ref> = i
    c
  }
```

→ `java.lang.Cloneable.IsInstanceOf(arg)`

similarly for `isInstanceOf`.

2. `C` is array of ghost. Call the static `IsInstanceArray(arg, rank)` on the ghost's type.

   TODO: what about `asInstanceOf[`*array of ghost*`]`

3. if a LHS below matches `C`, rewrite `C` as follows:

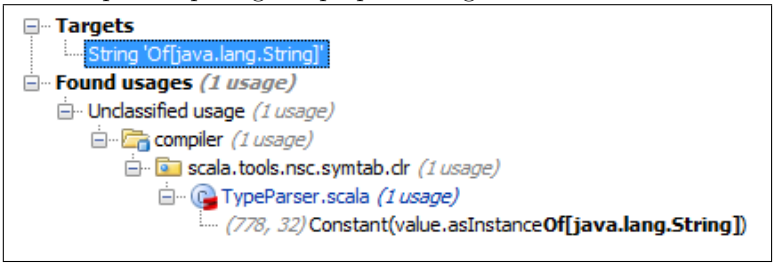
   - `java.lang.Object` → `System.Object`.
     Example: because `"abc".isInstanceOf[java.lang.Object]` evaluates to `true` on JVM, but won't on CLR unless reformulated as `"abc".isInstanceOf[System.Object]`
   - `java.lang.String` → `System.String`
   - All of the following into `System.Exception`:
     - `java.lang.Throwable`,
     - `java.lang.Error`,
     - `java.lang.Exception`,

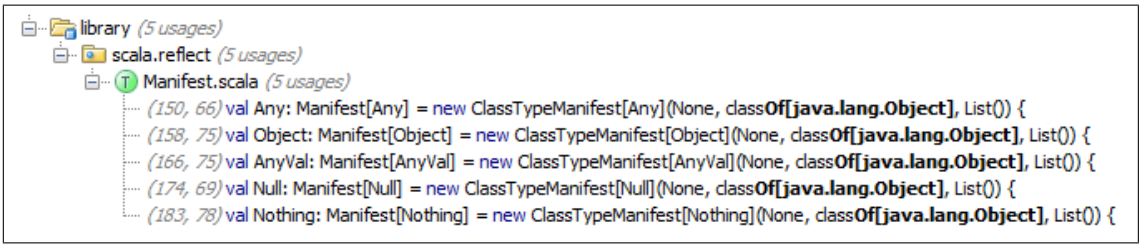
4. otherwise: leave as-is.

In contrast, `classOf[]` should be left as-is because it's special-cased anyway by the compiler.

Background: Those spots requiring rewriting are found with textual searches involving the FQN, like "`Of[java.lang.Exception]`" and so on for `java.lang.Error` and `java.lang.Throwable`. In contrast, the non-qualified name requires no rewriting, e.g., `instanceOf[Exception]` is left as-is. Similarly for "`Of[java.lang.String]`" and "`Of[java.lang.Object]`". Better yet, these two last should be updated in `trunk` to refer to `String` and `Object`, thus doing away with the need for preprocessing.

All the spots requiring the preprocessing discussed in this section are:





2 Fulfilling the `j.l.Object` contract without inheriting from `j.l.Object`

2.1 The problem: Illegal inheritance (two classes to extend from)

If we try to have all output classes inherit from IKVM's `java.lang.Object` we get in trouble. Say we leave unchanged:

```
trait ScalaObject extends java.lang.Object
```

Consequences:

```
scala\UninitializedFieldError.scala:24: error: illegal inheritance; superclass RuntimeException
 is not a subclass of the superclass Object
 of the mixin trait ScalaObject
final case class UninitializedFieldError(msg: String)
                                         ^
```

Before-translation, we have:

```
final case class UninitializedFieldError(msg: String)
          extends RuntimeException(msg) {
  def this(obj: Any) =
    this(if (null != obj) obj.toString() else "null")
}
```

In IKVM, the supertypes of `j.l.RuntimeException` (shown below) do not include `j.l.Object`, while the (original) definition of `ScalaObject` requires `j.l.Object` as superclass:

```
java.lang.RuntimeException
  java.lang.Exception
    java.lang.Throwable
      System.Exception
        System.Object
        System.Runtime.InteropServices._Exception
        System.Runtime.Serialization.ISerializable
      java.io.Serializable+__Interface
```

BTW, `ikvmc` does not face that problem because in the `.jar` there are two classes:

- `scala.UninitializedFieldError` with supertypes:

```
java.lang.RuntimeException
    java.lang.Exception
        java.lang.Throwable
```

```
            System.Exception
                System.Object
                System.Runtime.InteropServices._Exception
                System.Runtime.Serialization.ISerializable
            java.io.Serializable+__Interface
java.io.Serializable+__Interface
scala.Product
    scala.Equals
    scala.ScalaObject
```

- and `scala.UninitializedFieldError$` with supertypes:

```
scala.runtime.AbstractFunction1
    java.lang.Object
        System.Object
    scala.Function1
        scala.ScalaObject
java.io.Serializable+__Interface
System.Runtime.Serialization.IObjectReference
System.Runtime.Serialization.ISerializable
```

The "problem" with `ScalaObject` extending `System.Object` is that more rewritings are needed.

2.1.1 Let `ScalaObject` perform the detouring (doesn't compile)

```
trait ScalaObject extends System.Object {
       def toString = java.lang.Object.instancehelper_toString(this)
       // clone() and finalize() left out on purpose
       def getClass() = java.lang.Object.instancehelper_getClass(this)
       def hashCode() = java.lang.Object.instancehelper_hashCode(this)
       def equals(obj: Any) = java.lang.Object.instancehelper_equals(this, obj.asInstanceOf[AnyRef])
       def notify() { java.lang.Object.instancehelper_notify(this) }
       def notifyAll() { java.lang.Object.instancehelper_notifyAll(this) }
       def wait(timeout: Long) { java.lang.Object.instancehelper_wait(this, timeout) }
       def wait(timeout: Long, nanos: Int) { java.lang.Object.instancehelper_wait(this, timeout, nanos) }
       def wait() { java.lang.Object.instancehelper_wait(this) }
}
```

2.1.2 Let an implicit + view detour (doesn't cope with overrides)

A view from `j.l.String` → `S.String` works because no overrides can be made of `j.l.String`. The `j.l.Object` → `S.Object` view couldn't cope with overriding of non-final methods: a subclass with "`override def toString = ...`" results in errors like "nothing to override".

2.1.3 Let `ScalaObject` check fulfillment of the `j.l.Object` contract

The problem with the approach below is that it demands an implementation of each method to be provided, even if the method is never called. A post-`GenICode` phase could fill in those implementations.

```
trait ScalaObject extends System.Object {
  def toString : String
```

```
namespace System
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.AutoDual)]
    [Serializable]
    public class Object
    {
        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
        public Object();

        public virtual string ToString();
        public virtual bool Equals(object obj);
        public static bool Equals(object objA, object objB);

        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
        public static bool ReferenceEquals(object objA, object objB);

        public virtual int GetHashCode();

        [MethodImpl(MethodImplOptions.InternalCall)]
        public Type GetType();

        [ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
        ~Object();

        [MethodImpl(MethodImplOptions.InternalCall)]
        protected object MemberwiseClone();
    }
}
```

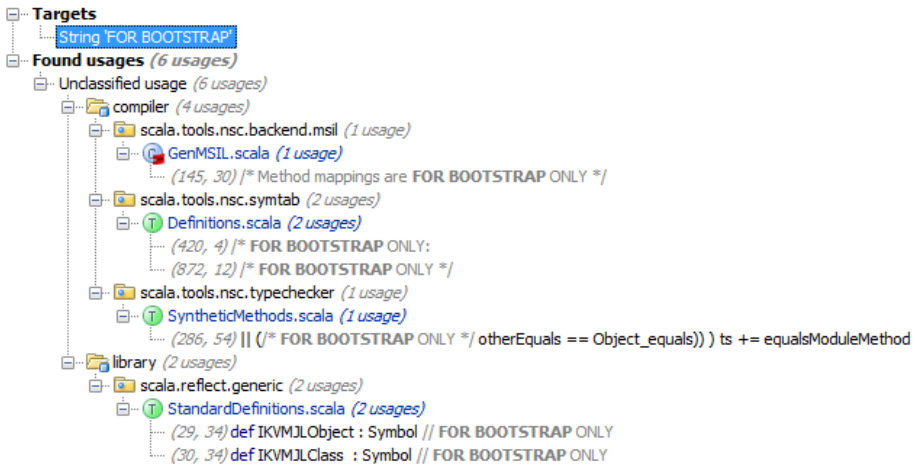
Figure 5: The Object contract on .NET

```
  // clone() and finalize() left out on purpose
  def getClass() : java.lang.Class
  def hashCode() : Int
  def equals(obj: Any) : Boolean
  def notify() : Unit
  def notifyAll() : Unit
  def wait(timeout: Long) : Unit
  // def wait(timeout: Long, nanos: Int) : Unit
  def wait() : Unit
}
```

To recap, the Object contract on .NET is shown in Figure 5 on p. 14.

2.2 Part of the solution: entering method symbols for the `j.l.Object` contract, replacing invocations afterwards

This subsection describes an interim solution, whose building blocks will be reused in the upcomging JDK-to-IKVM preprocessor. The current prototype allows validating early on the conversion recipe.



`Definitions.scala` enters method symbols in `ObjectClass` for the `j.l.Object` contract. Taken by itself, this allows compiling programs that are neither pure `forJVM` nor `forMSIL`. Unlike the bytecode that `ikvmc` would emit, many features "for interoperability" are missing.

This subsection is titled "Part of the solution . . . " because there's more to it (please jump to Sec. 2.3 on p. 16 for details). An excerpt from that discussion:

> *It's not enough to invoke IKVM's `instancehelper_toString(receiver)` instead of `receiver.toString` because the actual class of `receiver` may override `toString`. Without a `ToString` co-override in that actual class (IKVM's `instancehelpers` invoke such co-overrides) the original behavior won't be preserved. We don't remove the original `toString()` override (`ikvmc` does not remove it either).*

2.2.1 `Definitions.scala` part of the story

First, stubs for methods are entered:

```
if (forMSIL) {

  /* FOR BOOTSTRAP ONLY */
  . . .
  // additional methods of Object
  newMethod(ObjectClass, "clone", List(), AnyRefClass.typeConstructor)
  // wait in Java returns void, on .NET Wait returns boolean. by putting
  // 'booltype' the compiler adds a 'drop' after calling wait.
  newMethod(ObjectClass, "wait", List(), unitType)
  newMethod(ObjectClass, "wait", List(longType), unitType)
  newMethod(ObjectClass, "notify", List(), unitType)
  newMethod(ObjectClass, "notifyAll", List(), unitType)
  newMethod(ObjectClass, "getClass", List(), IKVMJLClass.tpe) // ClassClass.tpe
  newMethod(ObjectClass, "toString", List(), StringClass.tpe)
  newMethod(ObjectClass, "hashCode", List(), intType)
  newMethod(ObjectClass, "equals", anyparam, booltype)
  newMethod(ObjectClass, "finalize", List(), unitType)
}
```

With that, the following symbol lookups won't fail:

```
/* FOR BOOTSTRAP ONLY:
*  The following Object_blabla are used only in SourcelessComments.
*  Actually, Object_equals IS USED in GenICode, RefChecks, and SyntheticMethods;
*       and  Object_hashCode and Object_toString in SyntheticMethods. */

. . .

   def Object_getClass = getMember(ObjectClass, nme.getClass_)
   def Object_clone    = getMember(ObjectClass, nme.clone_)
   def Object_finalize = getMember(ObjectClass, nme.finalize_)
   def Object_notify   = getMember(ObjectClass, nme.notify_)
   def Object_notifyAll = getMember(ObjectClass, nme.notifyAll_)
   def Object_equals   = getMember(ObjectClass, nme.equals_)
   def Object_hashCode = getMember(ObjectClass, nme.hashCode_)
   def Object_toString = getMember(ObjectClass, nme.toString_)
```

2.2.2 `GenMSIL` part of the story

All `mapMethod` overloads have in common that they receive `class symbol, name` params. One overload takes additionally `msil-type, new-name`. Based on that, further params for the next overload (`array of param types (tpe's)`) can be computed. Finally, the most complete overload sports in addition `newParamTypes: Array[MsilType]` params.

```
mapMethod(JOBJECT, "getClass"    , jEmpty  , MIKVMJLObject , "instancehelper_getClass", mObject1)
mapMethod(JOBJECT, nme.hashCode_ , jEmpty  , MIKVMJLObject , "instancehelper_hashCode", mObject1)
mapMethod(JOBJECT, nme.equals_   , jObject1, MIKVMJLObject , "instancehelper_equals"  , mObject2)
mapMethod(JOBJECT, nme.clone_              , MOBJECT       , "MemberwiseClone"                 )
mapMethod(JOBJECT, nme.toString_ , jEmpty  , MIKVMJLObject , "instancehelper_toString", mObject1)
mapMethod(JOBJECT, nme.notify_   , jEmpty  , MMONITOR      , "Pulse"                 , mObject1)
mapMethod(JOBJECT, nme.notifyAll_, jEmpty  , MMONITOR      , "PulseAll"              , mObject1)
mapMethod(JOBJECT, nme.wait_     , jEmpty  , MIKVMJLObject , "instancehelper_wait"    , mObject1)
mapMethod(JOBJECT, nme.wait_     , jLong1  , MIKVMJLObject , "instancehelper_wait"    , Array(MOBJECT, MLONG))
// mapMethod(JOBJECT, nme.wait_     , Array(JLONG.tpe, JINT.tpe), MIKVMJLObject , "instancehelper_wait"    , Ar
mapMethod(JOBJECT, nme.finalize_           , MOBJECT       , "Finalize")
```

2.3 Method co-overrides for the `Object` contract

It's not enough to invoke IKVM's `instancehelper_toString(receiver)` instead of `receiver.toString` because the actual class of `receiver` may override `toString`. Without a `ToString` *co-override* in that actual class (IKVM's `instancehelper`s invoke such co-overrides) the original behavior won't be preserved.

At the end of `GenMSIL`'s `genClass(IClass)` now we have:

```
  addCoOverride(tBuilder, iclass, "hashCode", "GetHashCode")
  addCoOverride(tBuilder, iclass, "toString", "ToString")
  addCoOverride(tBuilder, iclass, "finalize", "Finalize")
  addCoOverrideEquals(tBuilder, iclass)
```

which all end up invoking:

```
private[GenMSIL] def addCoOverride(tBuilder: TypeBuilder, iclass: IClass,
                                   jdkName: String, dotnetName: String,
                                   tpeList: List[Type]) { /* FOR BOOTSTRAP ONLY */
  val mOpt = iclass.methods.find ( m =>
```

```
    (m.symbol.name.toString() == jdkName) && (m.symbol.tpe.paramTypes == tpeList)
  )
  val hasOverride = mOpt.isDefined
  if (hasOverride) {
    val m = mOpt.get
    val msym = m.symbol
    val targetMethod = methods(msym).asInstanceOf[MethodBuilder]
    if (!targetMethod.IsAbstract) {
      val attrs: Short = (MethodAttributes.Public | MethodAttributes.Virtual |
                          MethodAttributes.HideBySig).toShort
      var mParamTypes : Array[MsilType] = msilParamTypes(msym)
      val coMethod = tBuilder.DefineMethod(dotnetName, attrs, targetMethod.ReturnType, mParamTypes)
      val coCode = coMethod.GetILGenerator()
      coCode.Emit(OpCodes.Ldarg_0)
      for (i <- 0.until(mParamTypes.length)) {
        coMethod.DefineParameter(i, ParameterAttributes.None, msilName(m.params(i).sym))
        loadArg(coCode, false)(i+1)
      }
      coCode.Emit(OpCodes.Call, targetMethod)
      coCode.Emit(OpCodes.Ret)
    }
  }
}
```

3 Adding missing overrides that JDK's `Object` used to provide

Some Scala library traits (e.g., `scala.Equals`) and some IKVM interfaces (e.g., `java.lang.CharSequence.__Interface`)) declare method signatures whose implementation is provided by JDK's `j.l.Object` (i.e., the non-final methods there). However, under IKVM, our classes don't get those implementations (as required to support those interfaces) because our classes don't inherit from `j.l.Object` but from `System.Object`.

Therefore, in those cases where an ancestor requires an implementation of `equals`, `toString`, or `hashCode`, and (a) no implementation is added by `SyntheticMethods` so far; (b) nor provided by any ancestor; and (c) this class is concrete; then add an implementation to invoke the `System.Object` counterpart.

3.1 Backround

First, some background about `SyntheticMethods`:

Overrides of `equals(Any)`, `toString()`, and `hashCode()` are added (as `Trees`) to case classes by `SyntheticMethods`. If we forget to co-override those (or any others similarly synthesized) then (a) the resulting program will have a different behavior on .NET, but (b) type checking won't complain. Quoting from `SyntheticMethods.scala`:

- *`productArity`, `element` implementations added to case classes*
- *`equals`, `hashCode` and `toString` methods are added to case classes, unless they are defined in the class or a baseclass different from `java.lang.Object`*
- *`toString` method is added to case objects, unless they are defined in the class or a baseclass different from `java.lang.Object`*

3.2 Mechanics: `SyntheticMethods` now adds a few `DefDefs` more

The starting point are the following invocations in `addSyntheticMethods`. For all the details, see Listing 3.

```
addMissingObjectContract(Object_equals)
addMissingObjectContract(Object_toString)
addMissingObjectContract(Object_hashCode)
addMissingIEnumerableContract()
addMissingIDisposableContract()
```

3.3 What-if missing: diagnosing with `peverify`'s help

Without adding those missing methods, we get cryptic "Type load failed" errors. For example,

```
Z:\scalaproj\mscor\sn3>peverify scalalib.dll /TRANSPARENT

Microsoft (R) .NET Framework PE Verifier. Version 4.0.30319.1
Copyright (c) Microsoft Corporation. All rights reserved.

[token 0x02000ABD] Type load failed.
[token 0x02000AC2] Type load failed.
[token 0x02000ACA] Type load failed.
3 Error(s) Verifying scalalib.dll
```

`ildasm` shows that `[token 0x02000ABD]` belongs to `JUComparatorDetour`, which implements `java.util.Comparator` but in the MSIL version does not implement `equals`, which is required by the IKVM version of `java.util.Comparator`.

4 Adding missing overrides for "*implied interfaces*"

This section explains the mechanics of a rewriting whose usefulness is covered in Sec. 7.1 (that's a forward reference, but the mechanics fits with the preceding discussion of `SyntheticMehtods`).

4.1 Adding missing `IEnumerable` overrides

Recipe: the `GetEnumerator()` override for an `Iterable` should return `new ikvm.lang.IterableEnumerator(this)`, that for a `Map` should return `new ikvm.lang.MapEnumerator(this)`.

```
def msilOverrideMethod(name: Name, targetmsym: Symbol, rhs: Tree): Tree = {
  val paramtypes = targetmsym.tpe.paramTypes
  val method     = syntheticMethod(
    name, 0, makeTypeConstructor(paramtypes, targetmsym.tpe.resultType)
  )
  typer typed { DEF(method) === rhs }
}

def addMissingIEnumerableContract() { /* FOR BOOTSTRAP ONLY */
  if (clazz.isAbstractClass) return
  val msym = getMember(IKVMSCIEnumerable, "GetEnumerator")
```

```
  if (!inheritsAbstrDecl(msym)) return
  if (definesOrInheritsImplOtherThanAnyRef(msym)) return
  val jlIterable = definitions.getClass("java.lang.Iterable")
  val argss = List(List(This(clazz)))
  val rhs : Tree = if (clazz.info.baseClasses contains jlIterable) {
    val ikvmIterableEnumerator = definitions.getClass("ikvm.lang.IterableEnumerator")
    New(TypeTree(ikvmIterableEnumerator.tpe), argss)
  } else {
    val ikvmMapEnumerator = definitions.getClass("ikvm.lang.MapEnumerator")
    New(TypeTree(ikvmMapEnumerator.tpe), argss)
  }
  ts += msilOverrideMethod("GetEnumerator", msym, rhs)
}
```

4.2 Adding missing `IDisposable` overrides

Long story short: IKVM's `java.io.Closeable` *does not* extend `System.IDisposable`. However, `java.io` and `java.nio` classes do, that's why `ikvmc` adds a detouring method from `IDisposable`'s `Dispose()` to `Closeable`'s `close()`.

```
// similar to msilObjectMethod, expect that here we invoke (1) on this a (2) jdk method
def msilDetourMethod(classWithTarget: Symbol, jdkName: Name, dotnetName: Name): Tree = {
  val target    = getMember(classWithTarget, jdkName)
  val paramtypes = target.tpe.paramTypes
  val method    = syntheticMethod(
    dotnetName, 0, makeTypeConstructor(paramtypes, target.tpe.resultType)
  )
  val toAdd = typer typed {
    DEF(method) === {
      val thisRef: Tree = Select(This(clazz), target)
      Apply( thisRef, method ARGNAMES )
    }
  }
  toAdd
}

def addMissingIDisposableContract() { /* FOR BOOTSTRAP ONLY */
  if (clazz.isAbstractClass) return
  val msym = getMember(IKVMSCIDisposable, "Dispose")
  if (!inheritsAbstrDecl(msym)) return
  if (definesOrInheritsImplOtherThanAnyRef(msym)) return
  val jioCloseable = definitions.getClass("java.io.Closeable")
  if (clazz.info.baseClasses contains jioCloseable) {
    val rhs : Tree = msilDetourMethod(jioCloseable, "close", "Dispose")
    ts += rhs
  } else {
    scala.Console.println("could not addMissingIDisposableContract for " + clazz) // error
  }
}
```

5 Parser-level rewritings

What follows has 50% to do with learning how the parser works (will be useful for adding LINQ support), and 50% motivated by the desire to automate the JDK-to-IKVM conversion. The upcoming preprocessor will follow instead a

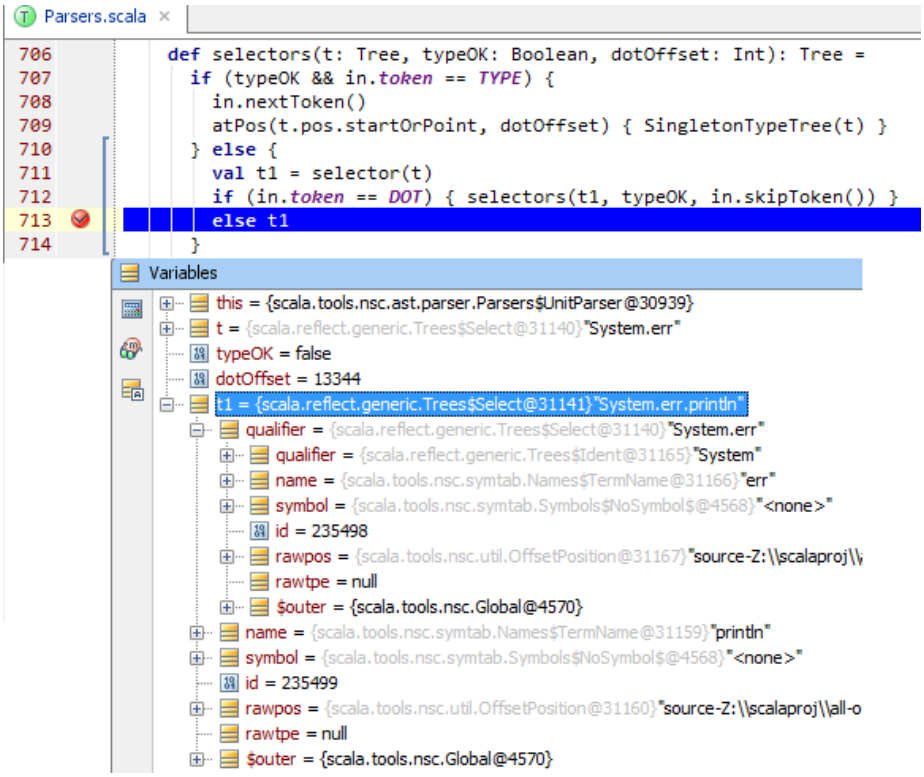


Figure 6: Rewriting "`System.`" selectors, Sec. 5.1

refactoring approach (i.e., after `typer` has run) as made possible by: `http://www.scala-refactoring.org/`

5.1 Rewriting "`System.`" selectors

Without qualification, selectors like `System.out` try to lookup `out` in .NET's `System`. Such occurrences should be qualified with `java.lang`, except for occurrences like:

```
System.Object, System.String, Systems.Collections.IEnumerable,
System.IDisposable, System.IComparable, System.Exception, System.Type
```

(which occur basically in `Predef.scala` and `LowPriorityImplicits.scala`).

In order to visualize what parse tree nodes we'll touch, the screen snap in Listing 6 on p. 20 is useful. The rewriting itself is performed as follows:

```
def selectors(t: Tree, typeOK: Boolean, dotOffset: Int): Tree = {

  class JLNameQualifier(unqn: Name) extends Transformer { /* FOR BOOTSTRAP ONLY */
    override def transform(tree: Tree): Tree = tree match {
      case Ident(unqn) =>
        atPos(tree.pos) { Select(Select(Ident(nme.java), nme.lang), unqn) }
      case _ => super.transform(tree)
    }
  }
```

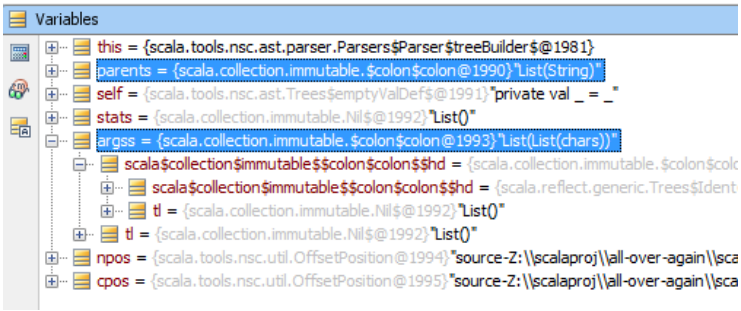
```
def qualifyWithJavaLang(tr: Tree) : Tree = { /* FOR BOOTSTRAP ONLY */
  val s = tr.toString
  def isJLSthgStaticAccess(unqClassName: String) =
    s.startsWith(unqClassName + ".") && s.charAt(unqClassName.length + 1).isLower
  for (unqClassName <- List("System", "Thread", "Integer", "Character", "Class")
       if (isJLSthgStaticAccess(unqClassName)) ) {
    val rwrttn = new JLNameQualifier(unqClassName) transform tr
    return rwrttn
  }
  if (s.startsWith("String.valueOf")) {
    // isLower rules out Predef occurrences such as System.Type
    new JLNameQualifier(nme.String) transform tr
  } else if (s.startsWith("Runtime.getRuntime")) {
    // isLower to make sure
    new JLNameQualifier("Runtime") transform tr
  } else tr
}
```

While useful, the transformation above is tricked by:

```
\scala\util\Properties.scala:55: error: value getenv is not a member of package System
  def envOrElse(name: String, alt: String)  = Option(System getenv name) getOrElse alt
                                                     ^
```

5.2 Rewriting `new String`

Before:



Transform:

```
/** Create positioned tree representing an object creation <new parents { stats }
 *  @param npos the position of the new
 *  @param cpos the position of the anonymous class starting with parents
 */
def makeNew(parents: List[Tree], self: ValDef, stats: List[Tree], argss: List[List[Tree]],
            npos: Position, cpos: Position): Tree =
  if (parents.isEmpty)
    makeNew(List(scalaAnyRefConstr), self, stats, argss, npos, cpos)
  else if (parents.tail.isEmpty && stats.isEmpty)
    atPos(npos union cpos) {
      if (parents.head.toString == "String") { /*- FOR BOOTSTRAP ONLY */
        val f = Select(gen.javaLangDot(nme.String), "newhelper")
        Apply(f, argss.head) /*- rewritten: java.lang.String.newhelper(<args>) */
      } else New(parents.head, argss)
    }
  else {
```

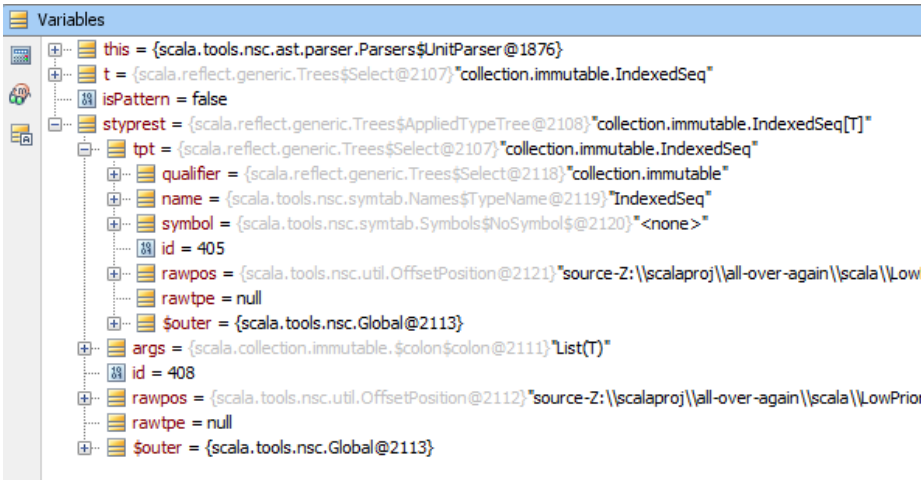
```
    val x = nme.ANON_CLASS_NAME.toTypeName
    atPos(npos union cpos) {
      . . .
    }
  }
```

5.3 Erasing type arguments to IKVM classes

5.3.1 Non-existentials

Start situation:

```
def simpleTypeRest(t: Tree, isPattern: Boolean): Tree =
  if (in.token == HASH) {
    val hashOffset = in.skipToken()
    val nameOffset = in.offset
    val name = ident(false)
    val sel = atPos(t.pos.startOrPoint, if (name == nme.ERROR) hashOffset else nameOffset) {
      SelectFromTypeTree(t, name.toTypeName)
    }
    simpleTypeRest(sel, isPattern)
  } else if (in.token == LBRACKET) {
    val styprest = simpleTypeRest(atPos(t.pos.startOrPoint) {
      AppliedTypeTree(t, typeArgs(isPattern, false)) }, isPattern)
    styprest
  } else {
    t
  }
```




```
/*- FOR BOOTSTRAP ONLY */
val jdkApplTypes = List("InheritableThreadLocal", "WeakHashMap", "LinkedBlockingQueue", "Callable",
  "java.util.concurrent.Future", "JClass", "java.lang.ref.Reference", "java.lang.ref.ReferenceQueue",
  "java.lang.ref.PhantomReference", "java.lang.ref.WeakReference", "java.lang.ref.SoftReference",
  "java.lang.Class", "ThreadLocal", "Stack[URL]", "Stack[Locator]")

def simpleTypeRest(t: Tree, isPattern: Boolean): Tree =
  if (in.token == HASH) {
    val hashOffset = in.skipToken()
    val nameOffset = in.offset
    val name = ident(false)
```

```
    val sel = atPos(t.pos.startOrPoint, if (name == nme.ERROR) hashOffset else nameOffset) {
      SelectFromTypeTree(t, name.toTypeName)
    }
    simpleTypeRest(sel, isPattern)
  } else if (in.token == LBRACKET) {
    val styprest = simpleTypeRest(atPos(t.pos.startOrPoint) { AppliedTypeTree(t, typeArgs(isPattern, false)) },
    styprest match { /*- FOR BOOTSTRAP ONLY */
      case AppliedTypeTree(tpt, args) if placeholderTypes.isEmpty =>
        if (jdkApplTypes exists (styprest.toString.startsWith(_) )) { // && !styprest.toString.contains("_")
          tpt
        } else styprest
      case _ => styprest
    }
  } else {
    t
  }
```

The guard "`if placeholderTypes.isEmpty`" is needed because the `Tree` for an existential type is half-way built above, i.e. the matched `AppliedTypeTree` in that case will in turn be contained in an `ExistentialTypeTree`. Thus, we can't just return the `tpt` element of the matched `AppliedTypeTree` in that case.

5.3.2 Existentials

Start situation:

```
281   def placeholderTypeBoundary(op: => Tree): Tree = {
282     val savedPlaceholderTypes = placeholderTypes
283     placeholderTypes = List()
284     var t = op
285     if (!placeholderTypes.isEmpty && t.isInstanceOf[AppliedTypeTree]) {
286       val expos = t.pos
287       ensureNonOverlapping(t, placeholderTypes)
288       t = atPos(expos) { ExistentialTypeTree(t, placeholderTypes.reverse) }
289       placeholderTypes = List()
290     }
291     placeholderTypes = placeholderTypes ::: savedPlaceholderTypes
292     t match { /* FOR BOOTSTRAP ONLY */
293       case ExistentialTypeTree(tpt, whereClauses) =>
294         if (jdkApplTypes exists (t.toString.startsWith(_) )) {
295           val res = tpt.asInstanceOf[AppliedTypeTree].tpt
296           res
297         } else t
298       case _ => t
299     }
300   }
```

```
this = {scala.tools.nsc.ast.parser.Parsers$UnitParser@2021}
op = {scala.tools.nsc.ast.parser.Parsers$Parser$$anonfun$annotType$1@2023}"<function0>"
savedPlaceholderTypes = {scala.collection.immutable.Nil$@1950}"List()"
t$1 = {scala.runtime.ObjectRef@2024}"JClass[_$1] forSome { \r\n  <synthetic> type _$1 >: _root_.scala.Nothing <: _root_.scala.Any\r\n}"
  elem = {scala.reflect.generic.Trees$ExistentialTypeTree@2033}"JClass[_$1] forSome { \r\n  <synthetic> type _$1 >: _root_.scala.Nothing <: _root_.sca
    tpt = {scala.reflect.generic.Trees$AppliedTypeTree@2025}"JClass[_$1]"
    whereClauses = {scala.collection.immutable.$colon$colon@2035}"List(<synthetic> type _$1 >: _root_.scala.Nothing <: _root_.scala.Any)"
    id = 48310
    rawpos = {scala.tools.nsc.util.OffsetPosition@2036}"source-Z:\\scalaproj\\all-over-again\\scala\\runtime\\MethodCache.scala,line-31,offset=1583"
    rawtpe = null
    $outer = {scala.tools.nsc.Global@1980}
tpt = {scala.reflect.generic.Trees$AppliedTypeTree@2025}"JClass[_$1]"
```

```
def placeholderTypeBoundary(op: => Tree): Tree = {
  val savedPlaceholderTypes = placeholderTypes
```

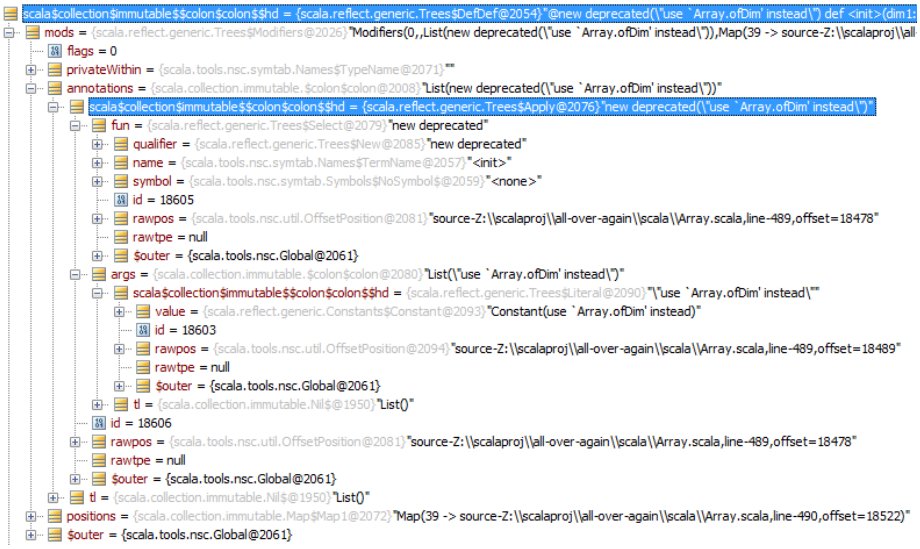


Figure 7: What method annotations looks like in the CST

```
  placeholderTypes = List()
  var t = op
  if (!placeholderTypes.isEmpty && t.isInstanceOf[AppliedTypeTree]) {
    val expos = t.pos
    ensureNonOverlapping(t, placeholderTypes)
    t = atPos(expos) { ExistentialTypeTree(t, placeholderTypes.reverse) }
    placeholderTypes = List()
  }
  placeholderTypes = placeholderTypes ::: savedPlaceholderTypes
  t match { /*- FOR BOOTSTRAP ONLY */
    case ExistentialTypeTree(tpt, whereClauses) =>
      if (jdkApplTypes exists (t.toString.startsWith(_) )) {
        val res = tpt.asInstanceOf[AppliedTypeTree].tpt
        res
      } else t
    case _ => t
  }
}
```

5.4 Ignoring `@throws` and `@serializable`

It's an `Apply` (a constructor invocation) kept in the `annotations` of the `DefDef` in question (Figure 7).

```
/** Annotations    ::= {'@' SimpleType {ArgumentExprs}}
 *  ConsrAnnotations ::= {'@' SimpleType ArgumentExprs}
 */
def annotations(skipNewLines: Boolean, requireOneArgList: Boolean): List[Tree] = {
  var annots = new ListBuffer[Tree]
  while (in.token == AT) {
    in.nextToken()
    /*- FOR BOOTSTRAP ONLY */
    val annTemp = annotationExpr(requireOneArgList)
    val annTempStr = annTemp.toString
```

```
    if (!annTempStr.startsWith("new throws") && !annTempStr.startsWith("new serializable")) {
      annots += annTemp
    }
    if (skipNewLines) newLineOpt()
  }
  annots.toList
}
```

6 Manual rewritings

6.1 `length` vs. `length()`

To get rid of this pesky issue:

```
scala\collection\immutable\List.scala:624: error: scala.this.Predef.augmentString(str).length of type Int does
    var pos = str.length()
                      ^
```

re-define in `scala.collection.immutable.StringLike`

```
trait StringLike[+Repr] extends IndexedSeqOptimized[Char, Repr] with Ordered[String] {
self =>
  . . .
  def length(): Int = toString.length /*- it used to be def length: Int = ... */
```

6.2 `Array.empty[T]`

```
val id = Int.unbox(classOf[Val] getMethod "id" invoke value)
```

→

```
val id = Int.unbox(classOf[Val].getMethod("id", new Array[java.lang.Class](0)).invoke(value,
new Array[Object](0)))
```

Make the class public, and the methods too. Otherwise has assembly visibility on .NET, and needs to be compiled into the same assembly as `scalalib.dll` (not always the case, e.g. if written in C#).

6.3 `ArrayRuntime` should be public

`ArrayRuntime` has its static fields not marked `public`, resulting in `assembly` visibility. When placed in `JavaFilesInScalaLibrary.dll`, those methods are not entered by `TypeParser` as they are not in the same assembly as that being compiled.

```
package scala.runtime;

/**
 * Methods on Java arrays
 */
class ArrayRuntime {
  static boolean[] cloneArray(boolean[] array) { return array.clone(); }
  static byte[] cloneArray(byte[] array) { return array.clone(); }
  static short[] cloneArray(short[] array) { return array.clone(); }
  static char[] cloneArray(char[] array) { return array.clone(); }
  static int[] cloneArray(int[] array) { return array.clone(); }
```

```
  static long[] cloneArray(long[] array) { return array.clone(); }
  static float[] cloneArray(float[] array) { return array.clone(); }
  static double[] cloneArray(double[] array) { return array.clone(); }
  static Object[] cloneArray(Object[] array) { return array.clone(); }
}
```

7 Preprocessor: Magic around interfaces

7.1 Implied interfaces

The IKVM-counterpart to a JDK interface may require "implementing in tandem" a .NET interface for interoperability (a so called *implied interface*). There are two such cases:

- `java.lang.Iterable`, `System.Collections.IEnumerable`
- `java.io.Closeable`, `System.IDisposable`

In the cases above, the IKVM version of the JDK interface *does not inherit* the .NET interface, but (some) classes implementing "the JDK interface" also require the .NET counterpart. That's why we can rely on a JDK-method implementation as target for detouring from the .NET counterpart.

Initially we tried to solve this with a view (Listing 4 on p. 36) but later realized a better solution was to add those missing methods implementations in `SyntheticMethods`, as described in Sec. 4.

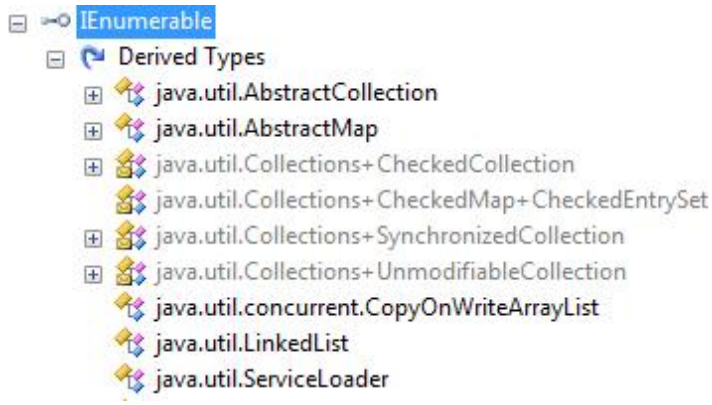
TODO: what about `__<>Dispose`. Or is it for `java.io.Closeable`?

7.1.1 `ikvmc` and the `IEnumerable` implied interface

`System.Collections.IEnumerable` is an *implied interface* of maps and lists. This is what IKVM's `map.xml` has to say about that extra interface:

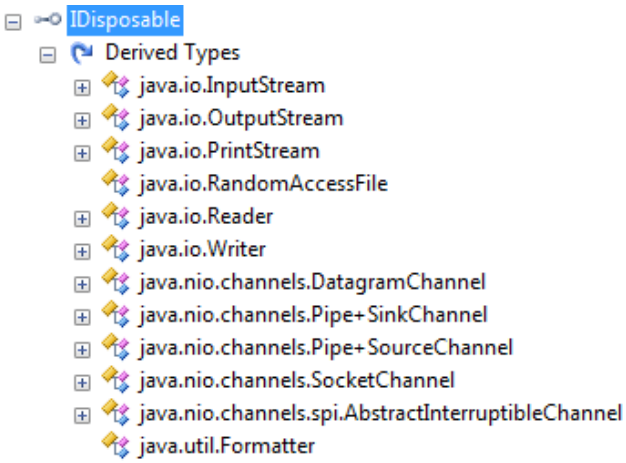
```
<class name="java.util.AbstractMap">
    <!-- NOTE the compiler will not magically make all Maps enumerable, so we manually implement IEnumerable her
    <implements class="cli.System.Collections.IEnumerable" />
    <method name="System.Collections.IEnumerable.GetEnumerator" sig="()Lcli.System.Collections.IEnumerator;" modi
        <override class="cli.System.Collections.IEnumerable" name="GetEnumerator" />
        <attribute type="IKVM.Attributes.HideFromJavaAttribute" sig="()V" />
        <body>
            <ldarg_0 />
            <newobj class="ikvm.lang.MapEnumerator" name="&lt;init&gt;" sig="(Ljava.util.Map;)V" />
            <ret />
        </body>
    </method>
    . . .
```

The above is for `java.util.AbstractMap`. Other "JDK" types with `IEnumerable` as extra interface are:



7.1.2 `ikvmc` and the `IDisposable` implied interface

Long story short: IKVM's `java.io.Closeable` *does not* extend `System.IDisposable`. However, `java.io` and `java.nio` classes do, that's why `ikvmc` adds a detouring method from `IDisposable`'s `Dispose()` to `Closeable`'s `close()`.




```
scala\tools\nsc\io\NullPrintStream.scala:13:
error: class NullPrintStream needs to be abstract, since method Dispose in trait IDisposable of type ()Unit is
class NullPrintStream extends PrintStream(new ByteArrayOutputStream()) { }
      ^
```

IKVM documentation on the above: `http://weblog.ikvm.net/default.aspx?date=2007-03-17`.

`ikvmc` deals with that extra interface as follows (in `TypeWrapper.cs`):

```
if (iface.Name == "java.io.Closeable"
      && !wrapper.ImplementsInterface(ClassLoaderWrapper.GetWrapperFromType(typeof(IDisposable))))
{
      typeBuilder.AddInterfaceImplementation(typeof(IDisposable));
      MethodBuilder mb = typeBuilder.DefineMethod("__<>Dispose", MethodAttributes.Private |
              MethodAttributes.Virtual | MethodAttributes.NewSlot | MethodAttributes.Final | MethodAttributes.Sp
              typeof(void), Type.EmptyTypes);
      typeBuilder.DefineMethodOverride(mb, typeof(IDisposable).GetMethod("Dispose"));
```

```
        ILGenerator ilgen = mb.GetILGenerator();
        ilgen.Emit(OpCodes.Ldarg_0);
        MethodWrapper mw = iface.GetMethodWrapper("close", "()V", false);
        mw.Link();
        mw.EmitCallvirt(ilgen);
        ilgen.Emit(OpCodes.Ret);
}
```

7.2 Extra interface (`j.l.Comparable`)

In a nutshell, we don't rewrite standalone occurrences of `java.lang.Comparable` to `System.IComparable`, nor add stubs. Instead, we use the view shown in Listing 5 on p. 36.

The idea followed by `ikvmc` consists in having the IKVM version of a JDK interface extend a .NET interface for interoperability (actually, the only such pair is `java.lang.Comparable` extends `System.IComparable`). Unlike a Scala trait, a .NET interface cannot provide concrete implementations for some methods in terms of others. Instead, `ikvmc` emits a method sub (for each method in the *extra interface*) that delegates to the user-provided counterpart. This process is called "completing the extra interface". When preprocessing Scala sources, only `CompareTo-compareTo` falls in this category.

For interoperability, `ikvmc` will favor (in explicit type references) `System.IComparable` (i.e., the extra interface) over `java.lang.Comparable` (i.e., the JDK interface that carries the extra baggage). That way, existing .NET binaries unaware about IKVM can interoperate while ignoring the more specific interface.

7.3 Ghost interfaces

A *ghost interface* is a JDK interface for which no same-name *interface* but a *struct* exists in IKVM. The term refers to: `java.lang.CharSequence`, `java.lang.Cloneable`, and `java.io.Serializable`. IKVM supports serialization as described in Sec. 9. Transformations at play:

- In general, standalone type references to `Cloneable` and `CharSequence` remain as-is (and thus denote a struct type after translation) but in the following contexts a certain rewriting applies:
  - implementing a ghost interface, rewrite as follows:
    - `implements Cloneable` (or in FQN form)
      → `implements java.lang.Cloneable.__Interface`
    - `implements CharSequence` (or in FQN form)
      → `implements java.lang.CharSequence.__Interface`
      In this latter case, remember to implement `toString()`. Not necessary on JDK (as it is inherited from `j.l.Object`) but after translation `System.Object` will not provide an implementation. IKVM includes `toString()` among the methods to implement in `java.lang.CharSequence.__Interface`
  - instantiating an anonymous class, rewrite as follows:

* `new Cloneable { ...}` (or in FQN form)
  → `new java.lang.Cloneable.__Interface { ...}`
* `new CharSequence { ...}` (or in FQN form)
  → `new CharSequenceAbstract { ...}`
  where

```
class CharSequenceAbstract extends java.lang.CharSequence.__Interface {
  override def toString() = java.lang.Object.instancehelper_toString(this)
}
```

- *Static accesses*: ghost types correspond to JDK interfaces with no static fields (FYI: the IKVM version of `j.l.CharSequence`, a valuetype, does have static methods for example the `==` operator overload mentioned below).
- *Instance method invocations (1 of 2)*: the JDK versions of `Serializable` and `Cloneable` define no methods of their own, so the issue of how to map calls to them does not arise. On the other hand, `CharSequence` has instance methods of its own, but textual occurrences of invocations can remain as-is, due to the view in Listing 6 on p. 36 (with it, before-translation invocations on `CharSequence` instance methods will find at runtime a conformant receiver).
- *Instance method invocations (2 of 2)*: Please notice that `==` and `!=` between two `CharSequence`s should bind to the following methods. This is taken care of by `TypeParser`.

```
public static bool operator ==(CharSequence sequence1, CharSequence sequence2)
{ return (sequence1.__<ref> == sequence2.__<ref>); }

public static bool operator !=(CharSequence sequence1, CharSequence sequence2)
{ return (sequence1.__<ref> != sequence2.__<ref>); }
```

8 Preprocessor: Exception handling

Before looking at the rewriting rules, let's spend a moment to review the context where those rules apply:

- All of `j.l.Throwable`, `j.l.Exception`, and `j.l.Error` "get mapped to" `System.Exception` (because there's a correspondence between `j.l.Throwable` and `S.Exception` methods, and because neither `j.l.Exception` nor `j.l.Error` add methods of their own).
- IKVM's `j.l.Throwable` is derived from `System.Exception` and thus does not conform to `j.l.Object`.

The detailed recipe appears in Secs. 8.1 to 8.3, the high-level view is:

- After translation, each catch clause declares an argument of (a subclass of) `System.Exception`. Depending on the before-translation type, the rewritten type will be:

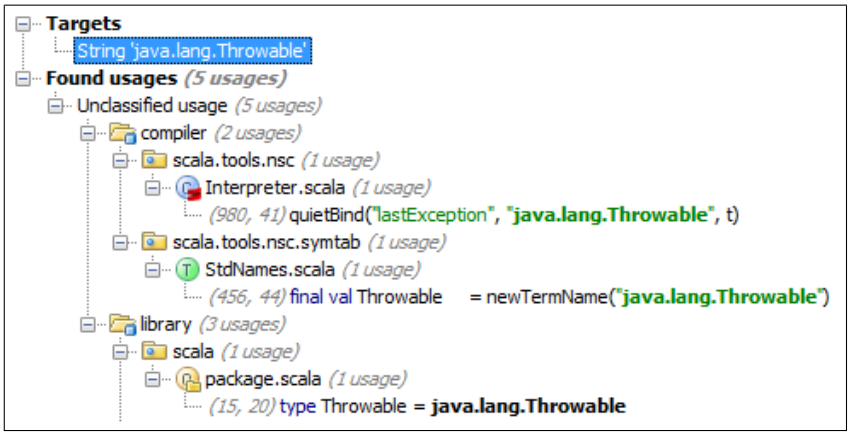


Figure 8: Textual occurrences of `java.lang.Throwable`, Sec. 8

  - `System.Exception` for `j.l.Throwable`, `j.l.Exception`, and `j.l.Error`, i.e., for Cases (1) and (2) below.
  - the same-name IKVM counterpart for all others, Case (3).

- In Case (1), `catch Throwable`, the "original catch block" may contain invocations to `Throwable` methods to be called through a `System.Exception` reference. `ikvmc` detours those invocations to `instancehelper`s in `Throwable` that check if the passed object subclasses `Throwable` and:
  - if so, (a) `callvirt` the method in question,
  - if not, calls either (b.1) the closest `System.Exception` equivalent; or (b.2) a static helper in IKVM's `java.lang.Throwable`.
- In Case (2), `catch Exception` or `catch Error`, a utility call is pre-pended to the output catch-body, to try to wrap the exception so that the wrapper conforms to the originally declared one (or rethrow it otherwise).
- In Case (3), the exception to catch is a proper subtype of `Exception` or `Error`. The rewriting is simpler because there's an IKVM counterpart for that type.

FYI:

- Textual occurrences of `java.lang.Throwable`, Figure 8
- Textual occurrences of `java.lang.Error`, Figure 9
- Textual occurrences of `java.lang.Exception`, Figure 10

8.1 Case (1) Originally `catch Throwable`

The output catch clause looks as follows (see also view in Listing 8 on p. 37):

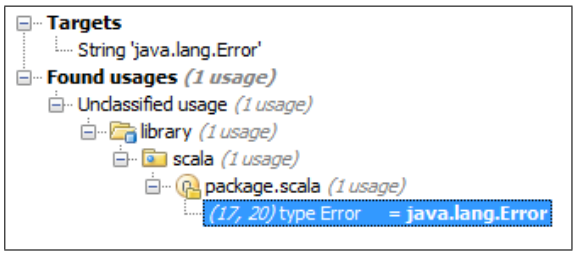


Figure 9: Textual occurrences of `java.lang.Error`, Sec. 8

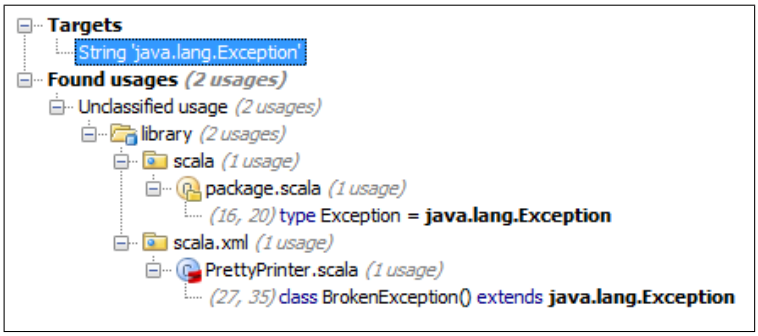


Figure 10: Textual occurrences of `java.lang.Exception`, Sec. 8

```
case exceptionArg : System.Exception =>
  System.Exception exception = Throwable.__<map>(exceptionArg, true);
  ... original catch block ...
  /*- in this block there are before-translation invocations
        on j.l.Throwable methods that after-translation will go through
        a System.Exception receiver. That's why view systemexceptionToJLThrowableDetour is needed. */
```

8.2 Case (2) Originally `catch Exception` or `catch Error`

The output catch clause looks as follows:

```
case exceptionArg : <ExceptionOrError> =>
  <ExceptionOrError> exception =
    Throwable.__<map>(exceptionArg,
                      typeof(<ExceptionOrError>), /*- TODO */
                      true).asInstanceOf[<ExceptionOrError>]
  if (exception eq null) { throw exceptionArg }
  ... original catch block ...
  /*- the view systemexceptionToJLThrowableDetour won't be used in this block */
```

8.3 Case (3) Otherwise

"Otherwise" means of course: originally a subclass of `Exception` or `Error` is caught.

```
case exceptionArg : <OriginalType> =>
  Throwable.__<map>(exceptionArg, true);
```

```
<OriginalType> exception = exceptionArg;
// the two stmts above can be simplified to just Throwable.__<map>(exception, true);
... original catch block ...
```

9 Preprocessor: Serialization

A class `SC` that either (a) extends `java.io.Serializable`, or (b) is annotated with `@serializable`; will be transformed as discussed below. FYI: in the `ikvmc` realization there are four conditions that preclude transformation, however none of them applies to Scala sources that compile in `forJVM` mode.

> `@serializable`-related methods are added (`readResolve()` during `SyntheticMethods`).

9.1 Case (A): `java.io.Externalizable`

If `SC` extends `java.io.Externalizable`, the recipe is TODO.

> TODO

9.2 Case (B): Base class is serializable

Otherwise, if `SC`'s base class is serializable (i.e., extends `Serializable` or is annotated `@serializable`):

- remove `j.l.Serializable` from the list of supported interfaces (or the `@serializable`) annotation) and add to that list `java.lang.Serializable.__Interface`
- annotate the output with `[Serializable]`
- add the following `readResolve()`:

```
def readResolve() : System.Object {
  return this /*- or is it better this.getClass.getField("MODULE$").get() */
}
```

- add a constructor taking `(SerializationInfo, StreamingContext)` to call the constructor with the same signature in the base class

```
override def this(info1 : SerializationInfo, context1 : StreamingContext) {
  super(info1, context1)
}
```

- `ikvmc` adds an implicit conversion operator to the output class. We have instead a single view in `Predef` (Listing 1 on p. 11).

9.3 Case (C): Otherwise

Otherwise, we assume the base class has a non-private default constructor:

- remove `j.l.Serializable` from the list of supported interfaces (or the `@serializable`) annotation) and add to that list `java.lang.Serializable.__Interface`

- annotate the output with `[Serializable]`
- add a `GetObjectData()` method as follows:

```
override def GetObjectData(info1 : SerializationInfo, context1 : StreamingContext) {
  ikvm.internal.Serialization.writeObject(this, info1);
}
```

- add a constructor as follows:

```
override def this(SerializationInfo info1, StreamingContext) {
  super()
  ikvm.internal.Serialization.readObject(this, info1);
}
```

- `ikvmc` adds an implicit conversion operator to the output class. We have instead a single view in `Predef` (Listing 1 on p. 11).
- add `readResolve()` method: TODO, where is it?

Listing 2: Detouring of `j.l.String`, Sec. 1.2

```
// Scala.NET code to be compiled against IKVM

implicit def sstringToJLStringDetour(s: System.String) = new JLStringDetour(s)

/*- need not extend IKVM's j.l.String (and in fact can't, that's sealed) */
class JLStringDetour(s: System.String) {
  // for each instance method in the original j.l.String, add a declaration as below
  def length() = java.lang.String.instancehelper_length(s)
  def isEmpty() = java.lang.String.instancehelper_isEmpty(s)
  def codePointAt(index: Int) = java.lang.String.instancehelper_codePointAt(s, index)
  def codePointBefore(index: Int) = java.lang.String.instancehelper_codePointBefore(s, index)
  def codePointCount(beginIndex: Int, endIndex: Int)
= java.lang.String.instancehelper_codePointCount(s, beginIndex, endIndex)
  def offsetByCodePoints(index: Int, codePointOffset: Int)
= java.lang.String.instancehelper_offsetByCodePoints(s, index, codePointOffset)
      // getChars(...) and other protected members can't be overriden, j.l.String is sealed both in JDK and in IK
  def getChars(srcBegin: Int, srcEnd: Int, dst: Array[Char], dstBegin: Int)
= java.lang.String.instancehelper_getChars(s, srcBegin, srcEnd, dst, dstBegin)
  def getBytes(charsetName: String) = java.lang.String.instancehelper_getBytes(s, charsetName)
  /*- we leave out j.l.Object instance methods (they are reweitten by the prototype) */
  def contentEquals(sb: java.lang.StringBuffer) = java.lang.String.instancehelper_contentEquals(s, sb)
  def contentEquals(cs: java.lang.CharSequence) = java.lang.String.instancehelper_contentEquals(s, cs)
  def equalsIgnoreCase(anotherString: String) = java.lang.String.instancehelper_equalsIgnoreCase(s, anotherStrin
  /*- we leave out compareTo (because there's already a view for Comparable) */
  /*- TODO: static methods and the CASE_INSENSITIVE_ORDER static field could be added for convenience, to avoid
  // instance method deriving from java.lang.CharSequence can be skipped, there's another view for them
  def compareToIgnoreCase(str: String) = java.lang.String.instancehelper_compareToIgnoreCase(s, str)
  def regionMatches(toffset: Int, other: String, ooffset: Int, len: Int) = java.lang.String.instancehelper_regio
  def regionMatches(ignoreCase: Boolean, toffset: Int, other: String, ooffset: Int, len: Int) = java.lang.String
  def startsWith(prefix: String, toffset: Int) = java.lang.String.instancehelper_startsWith(s, prefix, toffset)
  def startsWith(prefix: String) = java.lang.String.instancehelper_startsWith(s, prefix)
  // augmentString does it def endsWith(suffix: String) = java.lang.String.instancehelper_endsWith(s, suffix)
  def indexOf(ch: Int) = java.lang.String.instancehelper_indexOf(s, ch)
  . . .
  def matches(regex: String) = java.lang.String.instancehelper_matches(s, regex)
  // already in StringLike, i.e. augmentString takes the receiver there. def contains(cs: java.lang.CharSequence)
  def replaceFirst(regex: String, replacement: String) = java.lang.String.instancehelper_replaceFirst(s, regex,
  . . .
  def toUpperCase() = java.lang.String.instancehelper_toUpperCase(s)
  def trim() = java.lang.String.instancehelper_trim(s)
  def toCharArray() = java.lang.String.instancehelper_toCharArray(s)

}
```

Listing 3: See Sec. 3.2

```
def msilObjectMethod(jdkName: Name, dotnetName: Name): Tree = {
  val target     = getMember(ObjectClass, dotnetName)
  val paramtypes = target.tpe.paramTypes
  val method     = syntheticMethod(
    jdkName, 0, makeTypeConstructor(paramtypes, target.tpe.resultType)
  )
  val toAdd = typer typed {
    DEF(method) === {
      // how to build a 'forwarder to super' Tree: see for example overrideBridge in DeVirtualize
      val superRef: Tree = Select(Super(clazz, nme.EMPTY.toTypeName), target)
      /* if we invoked on this, i.e. with This(clazz) as qualifier, the co-override (i.e. ToString)
       *  would get into an endless loop as it invokes this.toString() */
      Apply( superRef, method ARGNAMES )
    }
  }
  toAdd
}

def inheritsAbstrDecl(meth: Symbol): Boolean = {
  val name = meth.name
  // does any parent other than AnyRef declare an abstract
  val bcs = clazz.info.baseClasses
  val abstrmems = for (bc <- bcs; patpe = bc.tpe; if patpe != ObjectClass.tpe;
       dsym = patpe.decl(name); d <- dsym.alternatives;
       if (d hasFlag DEFERRED) && (clazz.thisType.memberType(d) matches clazz.thisType.memberType(meth))
      ) yield d
  val res = !abstrmems.isEmpty && (abstrmems != List(NoSymbol))
  res
}

def definesOrInheritsImplOtherThanAnyRef(meth: Symbol): Boolean = {
  val sym = clazz.info nonPrivateMember meth.name
  val wtnss = sym.alternatives find { altsym =>
    altsym != meth &&
    !(altsym hasFlag DEFERRED) &&
    altsym.owner != ObjectClass &&
    (clazz.thisType.memberType(altsym) matches clazz.thisType.memberType(meth))
  }
  /* TODO if any other concrete class upstream in turn inheritsAbstrDecl(meth),
   * then once that class gets its addMissingObjectContract(msym)
   * it will have a public implementation of meth that clazz will inherit.
   * Therefore we need not add here again an override for it (moreover, with the same method body).
   * Behavior the same in both cases, but less code bloat. */
  wtnss.isDefined
}

def addMissingObjectContract(msym: Symbol) { /* FOR BOOTSTRAP ONLY */
  if (clazz.isAbstractClass) return
  val implReady = ts exists { case DefDef(_, ddname, _, ddvparamss, _, _)
                               => (ddname == msym.name) && (msym.paramss.head.length == ddvparamss.head.length)
                             case _ => false }
  if (implReady) return
  if (!inheritsAbstrDecl(msym)) return
  if (definesOrInheritsImplOtherThanAnyRef(msym)) return
  val t : Tree = msym.name match {
    case nme.hashCode_ => msilObjectMethod("hashCode", "GetHashCode")
    case nme.toString_ => msilObjectMethod("toString", "ToString")
    case nme.equals_ => msilObjectMethod("equals" , "Equals")
  }
  ts += t
}
```

Listing 4: (Failed) attempt to do the "implied interface" trick for `j.l.Iterable` and `j.io.Closeable`, Sec. 7.1

```
// Scala.NET code to be compiled against IKVM

implicit def java_lang_IterableToSystem_Collections_IEnumerable
(arg: java.lang.Iterable): System.Collections.IEnumerable =
  new System.Collections.IEnumerable {
    def GetEnumerator() = { new ikvm.lang.IterableEnumerator(arg) }
  }

implicit def java_io_CloseableToSystem_IDisposable
(arg: java.io.Closeable): System.IDisposable =
  new System.IDisposable {
    def Dispose() { arg.close() }
  }
```

Listing 5: Doing the "extra interface" trick, Sec. 7.2

```
// Scala.NET code to be compiled against IKVM

implicit def System_IComparableTojava_lang_Comparable
(argA: System.IComparable): java.lang.Comparable =
  new java.lang.Comparable {
    def compareTo(argB : Object) = { java.lang.Comparable.__Helper.compareTo(argA, argB) }
  }
```

Listing 6: `Predef` views to slim down the transformations for ghosts (`CharSequence` case), Sec. 7.3

```
// Scala.NET code to be compiled against IKVM

implicit def refToStructCharSequence
(i: java.lang.CharSequence.__Interface): java.lang.CharSequence = {
    val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
    c.__<ref> = i
    c
  }

implicit def sstringToStructCharSequence
(s: System.String): java.lang.CharSequence = {
    val c : java.lang.CharSequence = new java.lang.CharSequence() // default init
    c.__<ref> = s
    c
  }
```

Listing 7: `Predef` views to slim down the transformations for ghosts (`Cloneable` case), Sec. 7.3

```
// Scala.NET code to be compiled against IKVM

implicit def refToStructCloneable
(i: java.lang.Cloneable.__Interface): java.lang.Cloneable = {
    val c : java.lang.Cloneable = new java.lang.Cloneable() // default init
    c.__<ref> = i
    c
  }
```

Listing 8: Letting a `System.Exception` receive `j.l.Throwable` messages, Sec. 8

```
// view for Predef
// Scala.NET code to be compiled against IKVM

implicit def systemexceptionToJLThrowableDetour(e: System.Exception) = new JLThrowableDetour(e)

class JLThrowableDetour(e: System.Exception) /*- extends java.lang.Object */ {

    override def fillInStackTrace() =
      synchronized { java.lang.Throwable.instancehelper_fillInStackTrace(e) }

    override def getCause() =
      java.lang.Throwable.instancehelper_getCause(e)

    override def getLocalizedMessage() =
      java.lang.Throwable.instancehelper_getLocalizedMessage(e)

    override def getMessage() =
      java.lang.Throwable.instancehelper_getMessage(e)

    override def getStackTrace() =
      java.lang.Throwable.instancehelper_getStackTrace(e)

    override def initCause(cause: Throwable) =
      synchronized { java.lang.Throwable.instancehelper_initCause(e, cause) }

    override def printStackTrace() =
      java.lang.Throwable.instancehelper_printStackTrace(e)

    override def printStackTrace(s: PrintStream) =
      java.lang.Throwable.instancehelper_printStackTrace(e, s)

    override def printStackTrace(s: PrintWriter) =
      java.lang.Throwable.instancehelper_printStackTrace(e, s)

    override def setStackTrace(stackTrace: Array[StackTraceElement]) =
      java.lang.Throwable.instancehelper_setStackTrace(e, stackTrace)

    override def toString() =
      java.lang.Throwable.instancehelper_toString(e)

    /*- TODO there are also instancehelper_equals etc. */
}
```