How `ikvmc` works

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

September 16th, 2010

Abstract

In previous notes we reviewed the interplay between the IKVM library and the IKVM compiler (`ikvmc`). For bootstrapping, we want to feed the Scala.NET compiler with existing library and compiler sources, thus avoiding manual maintenance of a separate branch (automatically preprocessing those sources is ok). The IKVM library realizes JDK semantics provided that client programs follow certain usage conventions, which they always do because `ikvmc` performs program transformations on our behalf. When using Scala.NET rather than `ikvmc`, we have to preprocess the JDK-dependent Scala sources ourselves (thus making them IKVM-dependent). We describe how `ikvmc` works, save for those rewritings meaningful only at bytecode-level (e.g., conversion of exception tables from JVM to CLR). With that, formulating the remaining transformations at source-code level becomes possible, as well as developing a compiler plugin to perform them (to be covered in a follow-up write-up).

Structure of these notes

Sec. 1 contains an overview of the `ikvmc` phases that input classfiles go through before emerging as binary .NET assemblies. The intermediate representation comprises wrappers of different sorts. With that background, Secs. 2 and 3 zoom in on code generation, which is distributed over a number of classes. The details of transformations of interest are discussed starting with Sec. 4.

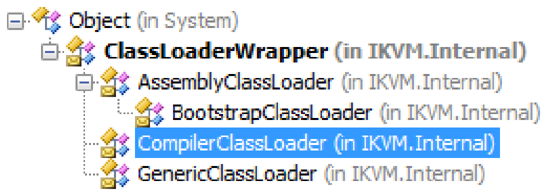
Role of `map.xml`

The XML dialect used in `map.xml` allows specifying transformations for `ikvmc` to apply when compiling OpenJDK classfiles (plus patches) into `IKVM.OpenJDK .dll`s (the "IKVM library"). It is true that most of these transformations are also necessary when IKVM-ifying non-JDK user programs (our use case of interest). However, in that setting, `ikvmc` parses the transformation patterns not from `map.xml` but from custom attributes in the IKVM library itself. Therefore, in this write-up we focus on that mode of operation of `ikvmc`.

Contents

1 Phases in `CompilerClassLoader`

The entry point to `ikvmc.exe` is `IkvmcCompiler.Main()` (in file `ikvmc/Compiler.cs`). Class `IkvmcCompiler` contains methods to parse command-line arguments and to populate the `List<CompilerOptions>` that a static method in another class, `CompilerClassLoader.Compile(optionsList)`, will receive. As shown below, `CompilerClassLoader` is a leaf class (with source file `ikvmc/CompilerClassLoader.cs`, the same file as for `CompilerOptions` and `StaticCompiler`).



```
Object (in System)
  ClassLoaderWrapper (in IKVM.Internal)
    AssemblyClassLoader (in IKVM.Internal)
      BootstrapClassLoader (in IKVM.Internal)
    CompilerClassLoader (in IKVM.Internal)
    GenericClassLoader (in IKVM.Internal)
```

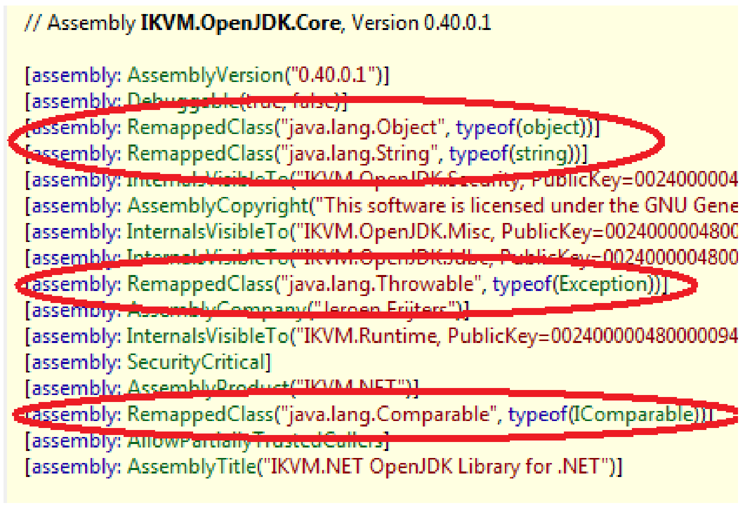
The static `CompilerClassLoader.Compile(optionsList)` stretches from lines 2440 to 2500 (not to be confused with the parameter-less `CompilerClassLoader.Compile()` which is the focus of Sec. 1.2). It iterates once over the `List<CompilerOptions>` creating a `CompilerClassLoader` for each, by means of another static method: `CompilerClassLoader.CreateCompiler(options, ref compiler, ref isCore)`

1.1 `CreateCompiler(options, ref compiler, ref isCore)`

This static method stretches between lines 2507 and 2857, doing the following:

- loads assemblies using `System.Reflection` (e.g., `StaticCompiler.runtimeAssembly` and those specified with `-reference`). Another distinguished assembly is `JVM.CoreAssembly` (e.g., `IKVM.OpenJDK.Core, Version=0.42.0.6, ...`);
- parses class files, by instantiating several classes defined in `runtime/ClassFile.cs`: `ClassFile` itself, each instance owning a `Field[]`, a `Method[]`, an `InnerClass[]`, and other stuff (e.g., the `object[] annotations` field, the `ConstantPoolItemClass[] interfaces` field, and the `ConstantPoolItem[] constantpool` field).
- some more fields in `CompilerOptions` are populated.
- the referenced assemblies are turned into `AssemblyClassLoaders` (for example, `IKVM.OpenJDK.Core` gets an `IKVM.Internal.BootstrapClassLoader`) and added to the created compiler (identified by "`loader`" below).

```
AssemblyClassLoader[] referencedAssemblies = new AssemblyClassLoader[references.Count];
for(int i = 0; i < references.Count; i++)
{
   referencedAssemblies[i] = AssemblyClassLoader.FromAssembly(references[i]);
}
loader = new CompilerClassLoader(referencedAssemblies,
                                 options, options.path, options.keyfilename, options.keycontainer,
                                 options.version, options.targetIsModule, options.assembly,
                                 h);  /*- h is ... what? */
loader.baseClasses = baseClasses;
loader.assemblyAnnotations = assemblyAnnotations;
loader.classesToCompile = new List<string>(h.Keys);
```



```
// Assembly IKVM.OpenJDK.Core, Version 0.40.0.1

[assembly: AssemblyVersion("0.40.0.1")]
[assembly: Debuggable(true, false)]
[assembly: RemappedClass("java.lang.Object", typeof(object))]
[assembly: RemappedClass("java.lang.String", typeof(string))]
[assembly: InternalsVisibleTo("IKVM.OpenJDK.Security, PublicKey=0024000004
[assembly: AssemblyCopyright("This software is licensed under the GNU Gene
[assembly: InternalsVisibleTo("IKVM.OpenJDK.Misc, PublicKey=002400000480
[assembly: InternalsVisibleTo("IKVM.OpenJDK.Jdbc, PublicKey=002400000480
[assembly: RemappedClass("java.lang.Throwable", typeof(Exception))]
[assembly: AssemblyCompany("Jeroen Frijters")]
[assembly: InternalsVisibleTo("IKVM.Runtime, PublicKey=002400000480000094
[assembly: SecurityCritical]
[assembly: AssemblyProduct("IKVM.NET")]
[assembly: RemappedClass("java.lang.Comparable", typeof(IComparable))]
[assembly: AllowPartiallyTrustedCallers]
[assembly: AssemblyTitle("IKVM.NET OpenJDK Library for .NET")]
```

Figure 1: `LoadRemappedTypes()`, Sec. 1

- the constructor invocation above ("`new CompilerClassLoader`") invokes that of `ClassLoaderWrapper`, which establishes some type mappings:

```
static ClassLoaderWrapper()
{
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.BOOLEAN.TypeAsTBD] = PrimitiveTypeWrapper.BOOLEAN;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.BYTE.TypeAsTBD] = PrimitiveTypeWrapper.BYTE;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.CHAR.TypeAsTBD] = PrimitiveTypeWrapper.CHAR;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.DOUBLE.TypeAsTBD] = PrimitiveTypeWrapper.DOUBLE;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.FLOAT.TypeAsTBD] = PrimitiveTypeWrapper.FLOAT;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.INT.TypeAsTBD] = PrimitiveTypeWrapper.INT;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.LONG.TypeAsTBD] = PrimitiveTypeWrapper.LONG;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.SHORT.TypeAsTBD] = PrimitiveTypeWrapper.SHORT;
   globalTypeToTypeWrapper[PrimitiveTypeWrapper.VOID.TypeAsTBD] = PrimitiveTypeWrapper.VOID;
   LoadRemappedTypes();
}
```

  However, `LoadRemappedTypes()` does nothing yet for as a result of that invocation because `coreAssembly == null`. Only near the end of `CompilerClassLoader.CreateCompiler()` will its body run (look for `if(JVM.CoreAssembly == null && !compilingCoreAssembly)`). The net effect grabbing some assembly attributes (Figure 1) to populate the following static field in `CompilerClassLoader`:

```
private static Dictionary<Type, string> remappedTypes = new Dictionary<Type, string>();
```

- finally, `CompilerClassLoader.CreateCompiler()` does the following before returning to `CompilerClassLoader.Compile()`:

```
if(!compilingCoreAssembly)
{
   FakeTypes.Load(JVM.CoreAssembly);
}
```

Afterwards, each of the compilers thus created keeps track of the others as `peerReferences`. The created `CompilerClassLoaders` are iterated three more times, invoking the following instance methods on each `compiler`:

- `EmitRemappedTypes2ndPass()`. Does nothing in the normal case, its body is guarded to detect `-remap:map.xml`.
- `Compile()`. Details in Sec. 1.2.
- `Save()`, which stretches from line 502 to 572. It completes by invoking `Save()` on an `IKVM.Reflection.Emit.AssemblyBuilder`, after doing a number of things summarized in Sec. 1.3.

1.2 `Compile()`

The instance method `CompilerClassLoader.Compile()` stretches between lines 2894 and 3067.

1.2.1 Type wrappers for all `classesToCompile` (lines 2913 to 2942)

All `classesToCompile` (a `List<string>`) in the current compiler are iterated over, to obtain a `TypeWrapper` for each. For our purposes, all `classesToCompile` are user-defined. That makes a difference as to how a type wrapper is obtained:

- For a user-defined class, the first time it is seen, the method acting as `TypeWrapper` factory is `CompilerClassLoader.GetTypeWrapperCompilerHook(string name)` (well, that factory method delegates to `ClassLoaderWrapper.DefineClass(ClassFile f, object protectionDomain)`, which in turn delegates yet one level deeper).
- For JDK classes, please try to get some inspiration by looking at the snippet in Listing 1. Additional examples are given in Sec. 1.3.1.

While preparing the type wrapper for a user-defined class, its superclasses are also "wrapped", which leads to wrapping `java.lang.Object` and thus the mechanism in the 2nd item above. Similarly, methods and fields are wrapped in `MethodWrappers` and `FieldWrappers` (Sec. 1.3.2).

The `TypeWrapper` thus found is tracked in the following field in `ClassLoaderWrapper` (the base class of the current compiler's class):

```
private Dictionary<string, TypeWrapper> types = new Dictionary<string, TypeWrapper>();
```

`TypeWrapper` has many tentacles (fields: Figure 2, properties: Figure 3, methods: Figure 4) even without considering its subclasses (Figure 5 on p. 9).

1.2.2 `AotTypeWrapper.Finish()`, line 2929

Nested in the previous subsection we should have discussed IL generation, given that it's invoked from there. However, a lot of relevant terminology remains to be introduced, therefore we postpone that all-important discussion to Secs. 2 and 3. For context, the snippet in Listing 3 on p. 14 shows where codegen takes place.
BTW, I figured out about codegen by placing breakpoints in `ContructorBuilder`'s and `MethodBuilder`'s `GetILGenerator()`.

Listing 1: See Sec. 1.2

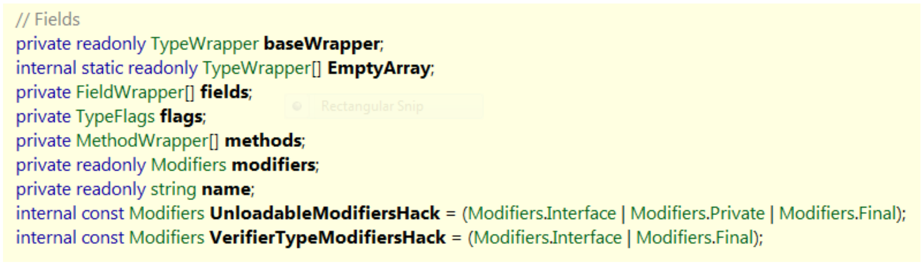
```
class CompiledTypeWrapper : TypeWrapper
{
   private readonly Type type;
   private TypeWrapper[] interfaces;
   private TypeWrapper[] innerclasses;
   private MethodInfo clinitMethod;
   private Modifiers reflectiveModifiers;

   internal static CompiledTypeWrapper newInstance(string name, Type type)
   {
      // TODO since ghost and remapped types can only exist in the core library assembly, we probably
      // should be able to remove the Type.IsDefined() tests in most cases
      if(type.IsValueType && AttributeHelper.IsGhostInterface(type))
      {
         return new CompiledGhostTypeWrapper(name, type);
      }
      else if(AttributeHelper.IsRemappedType(type))
      {
         return new CompiledRemappedTypeWrapper(name, type);
      }
      else
      {
         return new CompiledTypeWrapper(name, type);
      }
   }
   . . .
```

1.2.3 Finale (lines 2943 to 3066)

Before getting into the next big flurry of activity (next section), more work is done during the current invocation of `CompilerClassLoader.Compile()`:

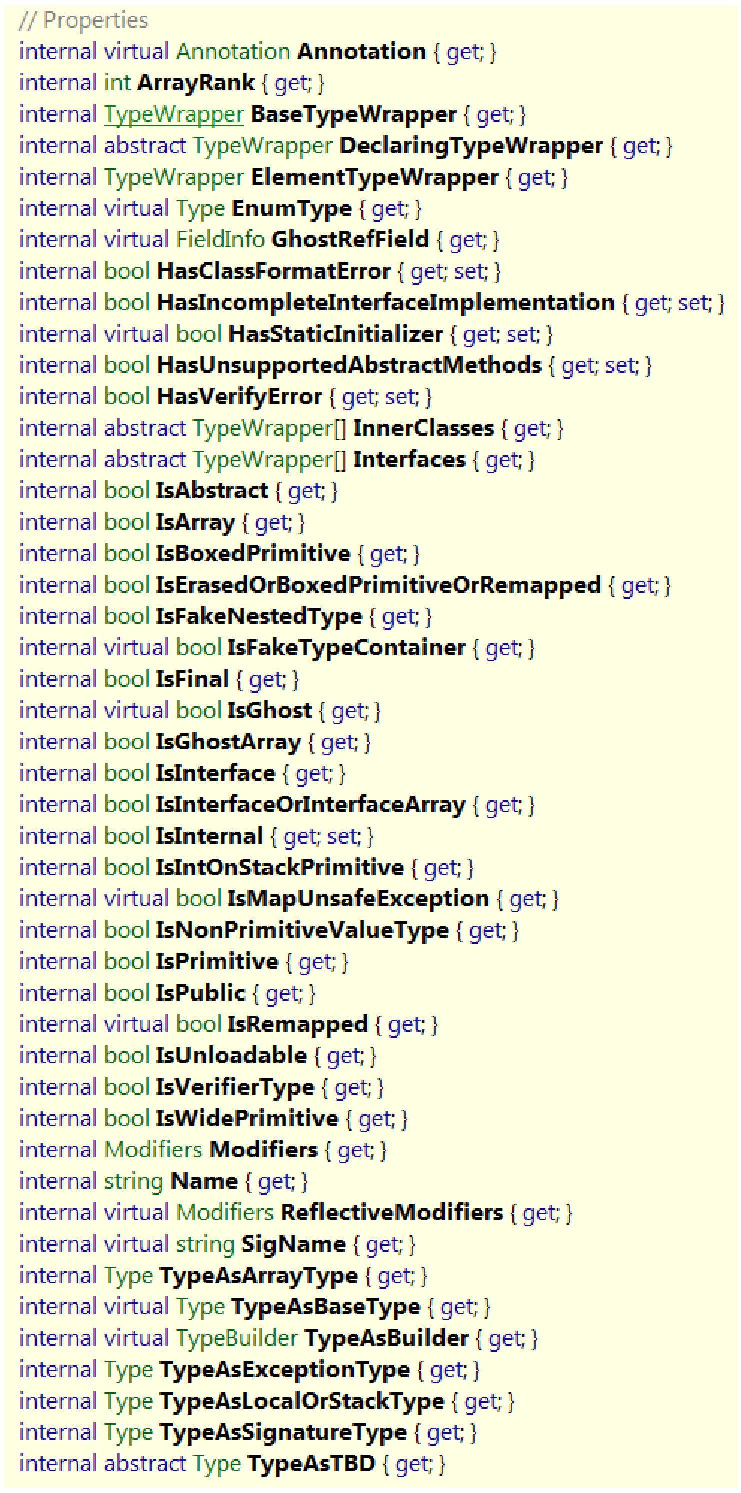
- lines 2943 to 2987, “`if(options.mainClass != null)`”
  - finds the main class, its main method (as a `MethodWrapper` and as a `MethodInfo`),
  - and calls

    ```
    SetMain(MethodInfo m, PEFileKinds target, Dictionary<string, string> props,
            bool noglobbing, Type apartmentAttributeType)
    ```



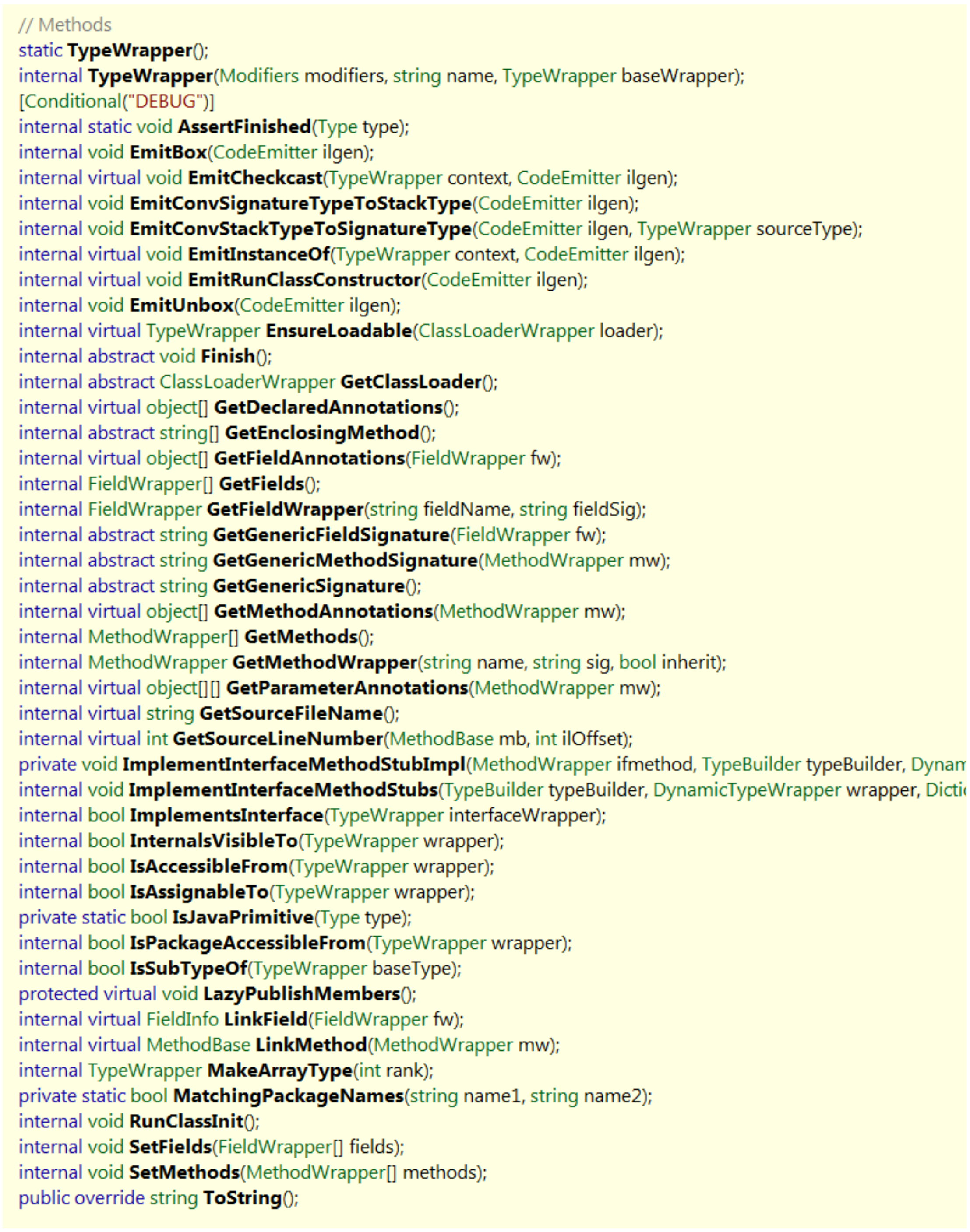
```
// Fields
private readonly TypeWrapper baseWrapper;
internal static readonly TypeWrapper[] EmptyArray;
private FieldWrapper[] fields;
private TypeFlags flags;
private MethodWrapper[] methods;
private readonly Modifiers modifiers;
private readonly string name;
internal const Modifiers UnloadableModifiersHack = (Modifiers.Interface | Modifiers.Private | Modifiers.Final);
internal const Modifiers VerifierTypeModifiersHack = (Modifiers.Interface | Modifiers.Final);
```

Figure 2: `TypeWrapper`’s fields, Sec. 1.2



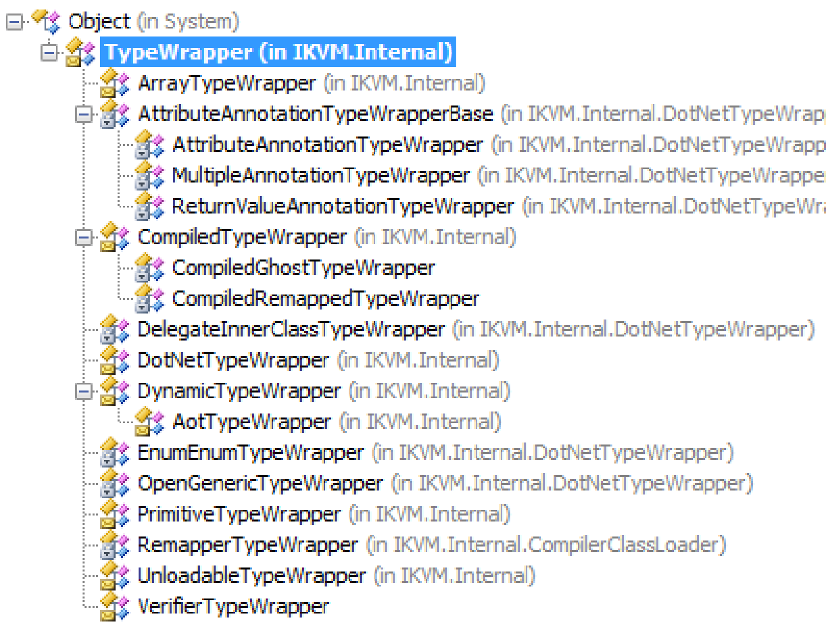
```
// Properties
internal virtual Annotation Annotation { get; }
internal int ArrayRank { get; }
internal TypeWrapper BaseTypeWrapper { get; }
internal abstract TypeWrapper DeclaringTypeWrapper { get; }
internal TypeWrapper ElementTypeWrapper { get; }
internal virtual Type EnumType { get; }
internal virtual FieldInfo GhostRefField { get; }
internal bool HasClassFormatError { get; set; }
internal bool HasIncompleteInterfaceImplementation { get; set; }
internal virtual bool HasStaticInitializer { get; set; }
internal bool HasUnsupportedAbstractMethods { get; set; }
internal bool HasVerifyError { get; set; }
internal abstract TypeWrapper[] InnerClasses { get; }
internal abstract TypeWrapper[] Interfaces { get; }
internal bool IsAbstract { get; }
internal bool IsArray { get; }
internal bool IsBoxedPrimitive { get; }
internal bool IsErasedOrBoxedPrimitiveOrRemapped { get; }
internal bool IsFakeNestedType { get; }
internal virtual bool IsFakeTypeContainer { get; }
internal bool IsFinal { get; }
internal virtual bool IsGhost { get; }
internal bool IsGhostArray { get; }
internal bool IsInterface { get; }
internal bool IsInterfaceOrInterfaceArray { get; }
internal bool IsInternal { get; set; }
internal bool IsIntOnStackPrimitive { get; }
internal virtual bool IsMapUnsafeException { get; }
internal bool IsNonPrimitiveValueType { get; }
internal bool IsPrimitive { get; }
internal bool IsPublic { get; }
internal virtual bool IsRemapped { get; }
internal bool IsUnloadable { get; }
internal bool IsVerifierType { get; }
internal bool IsWidePrimitive { get; }
internal Modifiers Modifiers { get; }
internal string Name { get; }
internal virtual Modifiers ReflectiveModifiers { get; }
internal virtual string SigName { get; }
internal Type TypeAsArrayType { get; }
internal virtual Type TypeAsBaseType { get; }
internal virtual TypeBuilder TypeAsBuilder { get; }
internal Type TypeAsExceptionType { get; }
internal Type TypeAsLocalOrStackType { get; }
internal Type TypeAsSignatureType { get; }
internal abstract Type TypeAsTBD { get; }
```

Figure 3: `TypeWrapper`'s properties, Sec. 1.2



```
// Methods
static TypeWrapper();
internal TypeWrapper(Modifiers modifiers, string name, TypeWrapper baseWrapper);
[Conditional("DEBUG")]
internal static void AssertFinished(Type type);
internal void EmitBox(CodeEmitter ilgen);
internal virtual void EmitCheckcast(TypeWrapper context, CodeEmitter ilgen);
internal void EmitConvSignatureTypeToStackType(CodeEmitter ilgen);
internal void EmitConvStackTypeToSignatureType(CodeEmitter ilgen, TypeWrapper sourceType);
internal virtual void EmitInstanceOf(TypeWrapper context, CodeEmitter ilgen);
internal virtual void EmitRunClassConstructor(CodeEmitter ilgen);
internal void EmitUnbox(CodeEmitter ilgen);
internal virtual TypeWrapper EnsureLoadable(ClassLoaderWrapper loader);
internal abstract void Finish();
internal abstract ClassLoaderWrapper GetClassLoader();
internal virtual object[] GetDeclaredAnnotations();
internal abstract string[] GetEnclosingMethod();
internal virtual object[] GetFieldAnnotations(FieldWrapper fw);
internal FieldWrapper[] GetFields();
internal FieldWrapper GetFieldWrapper(string fieldName, string fieldSig);
internal abstract string GetGenericFieldSignature(FieldWrapper fw);
internal abstract string GetGenericMethodSignature(MethodWrapper mw);
internal abstract string GetGenericSignature();
internal virtual object[] GetMethodAnnotations(MethodWrapper mw);
internal MethodWrapper[] GetMethods();
internal MethodWrapper GetMethodWrapper(string name, string sig, bool inherit);
internal virtual object[][] GetParameterAnnotations(MethodWrapper mw);
internal virtual string GetSourceFileName();
internal virtual int GetSourceLineNumber(MethodBase mb, int ilOffset);
private void ImplementInterfaceMethodStubImpl(MethodWrapper ifmethod, TypeBuilder typeBuilder, Dynam
internal void ImplementInterfaceMethodStubs(TypeBuilder typeBuilder, DynamicTypeWrapper wrapper, Dictio
internal bool ImplementsInterface(TypeWrapper interfaceWrapper);
internal bool InternalsVisibleTo(TypeWrapper wrapper);
internal bool IsAccessibleFrom(TypeWrapper wrapper);
internal bool IsAssignableTo(TypeWrapper wrapper);
private static bool IsJavaPrimitive(Type type);
internal bool IsPackageAccessibleFrom(TypeWrapper wrapper);
internal bool IsSubTypeOf(TypeWrapper baseType);
protected virtual void LazyPublishMembers();
internal virtual FieldInfo LinkField(FieldWrapper fw);
internal virtual MethodBase LinkMethod(MethodWrapper mw);
internal TypeWrapper MakeArrayType(int rank);
private static bool MatchingPackageNames(string name1, string name2);
internal void RunClassInit();
internal void SetFields(FieldWrapper[] fields);
internal void SetMethods(MethodWrapper[] methods);
public override string ToString();
```

Figure 4: `TypeWrapper`'s methods, Sec. 1.2



```
Object (in System)
  TypeWrapper (in IKVM.Internal)
    ArrayTypeWrapper (in IKVM.Internal)
    AttributeAnnotationTypeWrapperBase (in IKVM.Internal.DotNetTypeWrap
      AttributeAnnotationTypeWrapper (in IKVM.Internal.DotNetTypeWrapp
      MultipleAnnotationTypeWrapper (in IKVM.Internal.DotNetTypeWrappe
      ReturnValueAnnotationTypeWrapper (in IKVM.Internal.DotNetTypeWr
    CompiledTypeWrapper (in IKVM.Internal)
      CompiledGhostTypeWrapper
      CompiledRemappedTypeWrapper
    DelegateInnerClassTypeWrapper (in IKVM.Internal.DotNetTypeWrapper)
    DotNetTypeWrapper (in IKVM.Internal)
    DynamicTypeWrapper (in IKVM.Internal)
      AotTypeWrapper (in IKVM.Internal)
    EnumEnumTypeWrapper (in IKVM.Internal.DotNetTypeWrapper)
    OpenGenericTypeWrapper (in IKVM.Internal.DotNetTypeWrapper)
    PrimitiveTypeWrapper (in IKVM.Internal)
    RemapperTypeWrapper (in IKVM.Internal.CompilerClassLoader)
    UnloadableTypeWrapper (in IKVM.Internal)
    VerifierTypeWrapper
```

Figure 5: `TypeWrapper` hierarchy

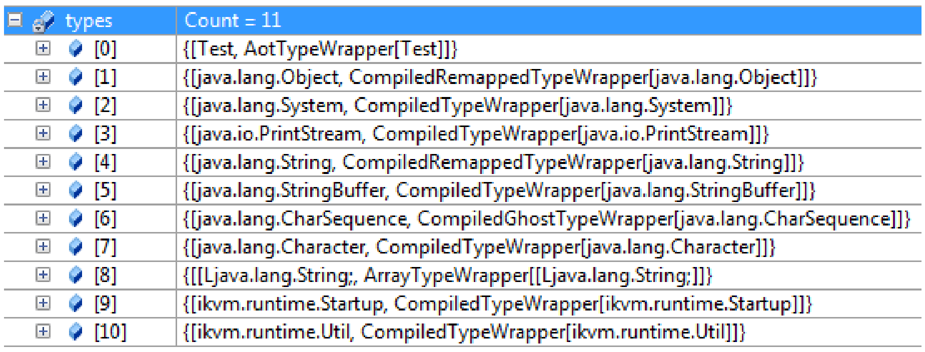
- lines 2988 to 3003, "`if(map != null)`". We don't run with a `map.xml` given on the command-line, thus this is a no-op.
- lines 3005 to 3012, adds resources if any.
- lines 3012 to 3017, deals with the `fileversion` command-line arg.
- lines 3018 to 3025, deals with the `assemblyAnnotations`.
- lines 3026 to 3064, "`if(options.classLoader != null)`", usually skipped.

1.3 `Save()`

The instance method `CompilerClassLoader.Save()` stretches over lines 502-572. We cover first the shape of objects reaching this method (Secs. 1.3.1 to 1.3.2) followed by the processing local to `Save()` proper (Sec. 1.3.3).

1.3.1 When which type wrapper

By the time `Save()` is reached all types, methods, fields, etc. will have been wrapped. For example, for the test program shown in Listing 2, `ikvmc` has wrapped as follows:



types	Count = 11
[0]	{[Test, AotTypeWrapper[Test]]}
[1]	{[java.lang.Object, CompiledRemappedTypeWrapper[java.lang.Object]]}
[2]	{[java.lang.System, CompiledTypeWrapper[java.lang.System]]}
[3]	{[java.io.PrintStream, CompiledTypeWrapper[java.io.PrintStream]]}
[4]	{[java.lang.String, CompiledRemappedTypeWrapper[java.lang.String]]}
[5]	{[java.lang.StringBuffer, CompiledTypeWrapper[java.lang.StringBuffer]]}
[6]	{[java.lang.CharSequence, CompiledGhostTypeWrapper[java.lang.CharSequence]]}
[7]	{[java.lang.Character, CompiledTypeWrapper[java.lang.Character]]}
[8]	{[[Ljava.lang.String;, ArrayTypeWrapper[[Ljava.lang.String;]]}
[9]	{[ikvm.runtime.Startup, CompiledTypeWrapper[ikvm.runtime.Startup]]}
[10]	{[ikvm.runtime.Util, CompiledTypeWrapper[ikvm.runtime.Util]]}

Summing up: both user-defined and JDK-provided types have been wrapped:

- wrappers for JDK-provided types include those for mapped, non-mapped, ghost, and array types (there are additional kinds of wrappers, Figure 5):
  - a mapped (JDK) type (e.g., `java.lang.Object`) gets a `CompiledRemappedTypeWrapper`
  - a non-mapped non-ghost JDK type (e.g., `java.io.PrintStream`) gets a `CompiledTypeWrapper`
  - a ghost (JDK) type (e.g., `java.lang.CharSequence`) gets a `CompiledGhostTypeWrapper`
  - an array type (e.g., `[java.lang.String`) gets an `ArrayTypeWrapper`, whose property `ElementTypeWrapper` points again to a type wrapper.
- a user-defined type is wrapped in an `AotTypeWrapper` (that class is a subclass of `DynamicTypeWrapper`). Question: `DynamicTypeWrapper` as opposed to `CompiledTypeWrapper`? Looks like classes loaded using reflection (later interpreted with help of `IKVM.Runtime.dll`) get a dynamic type wrapper, while those compiled with `ikvmc` get a compiled type wrapper.

1.3.2 When which member wrapper

Also by this time, `ikvmc` has performed some structural transformations that will appear unchanged in the output assembly. For example, the `AotTypeWrapper` for the `Test` class (Listing 2) contains four method definitions (in the `MethodWrapper[] methods` field):

- a `SmartConstructorMethodWrapper` with `RealName == ".ctor"` and a `PrimitiveTypeWrapper` (for `void`) as `ReturnType`. The `method` field of the `SmartConstructorMethodWrapper` points to an `IKVM.Reflection.Emit.ConstructorBuilder`. There's an `GetILGenerator()` for the contained `ConstructorBuilder`, which behaves differently from the same-name method for non-constructor (next item), thus it is reproduced below:

```
public ILGenerator GetILGenerator()
{
   return methodBuilder.GetILGenerator();
}
```

Listing 2: See Sec. 1.3

```
public class Test {

   public static void main(String[] args) {
      System.out.println("abc".hashCode());
   }

   public void testHash(Object arg) {
      System.out.println(arg.hashCode());
   }

   CharSequence toUpperCase(CharSequence seq) {
        StringBuffer buf = new StringBuffer();
        int len = seq.length();
        for(int i = 0; i < len; i++) {
          char c = seq.charAt(i);
          c = Character.toUpperCase(c);
          buf.append(c);
        }
        return buf;
   }

}
```

- the remaining three methods are represented with `SmartCallMethodWrappers`. Looking at `testHash`, its `parameterTypeWrappers[0]` is displayed in the debugger as `CompiledRemappedTypeWrapper[java.lang.Object]`, thus hinting that not all types in the method signature have been converted to their assembly-level counterparts (I guess only primitive ones have been converted?). And the `GetILGenerator()` for the contained `MethodBuilder`:

  ```
  public ILGenerator GetILGenerator()
  {
     if (ilgen == null)
     {
        ilgen = new ILGenerator(typeBuilder.ModuleBuilder);
     }
     return ilgen;
  }
  ```

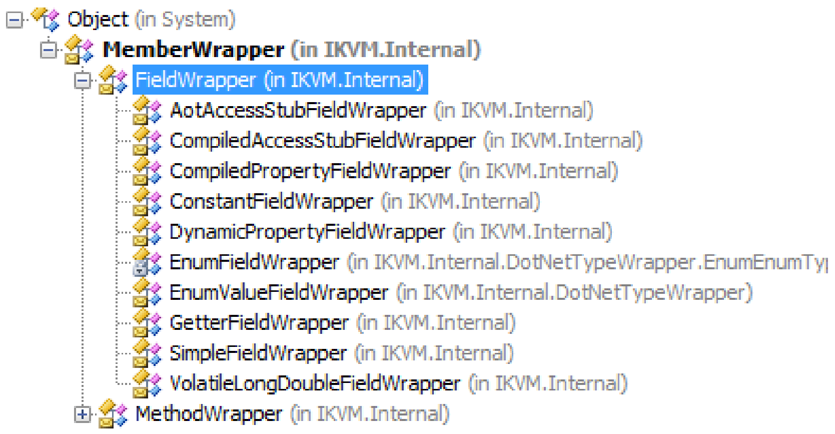
Member-related wrappers are classified as follows:

- Subclasses of `FieldWrapper` (Figure 6)
- Subclasses of `MethodWrapper` (Figure 7)

1.3.3 Adding entrypoint, module attributes, and resources

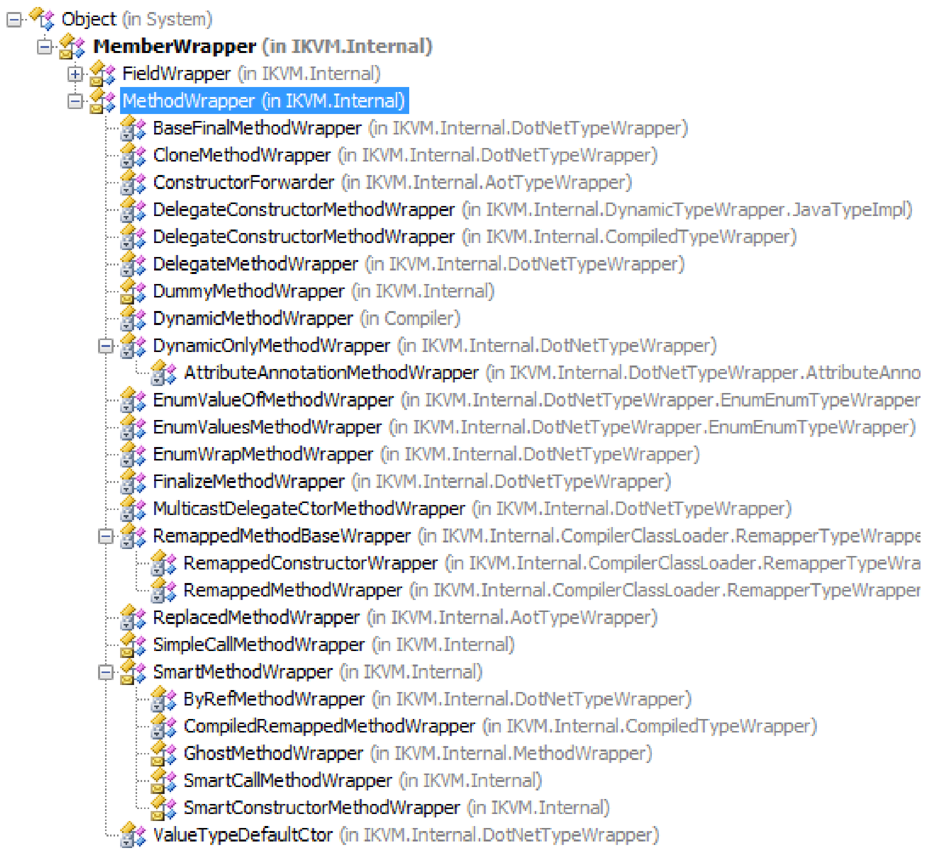
The highlights for lines 502–572 are:

- line 524, `mb.CreateGlobalFunctions()` creates the "real `main()`" that will invoke the user-provided `main` (`Test.main(string[] args)`) in the example (Figure 8)



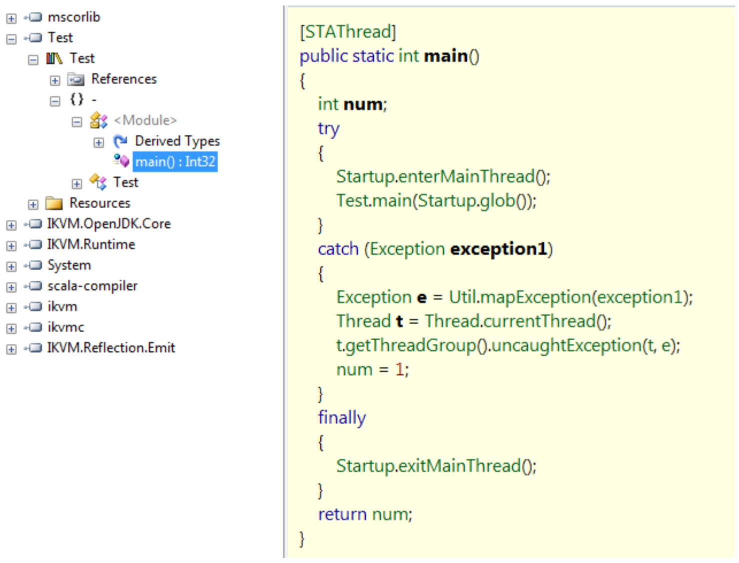
```
Object (in System)
  MemberWrapper (in IKVM.Internal)
    FieldWrapper (in IKVM.Internal)
      AotAccessStubFieldWrapper (in IKVM.Internal)
      CompiledAccessStubFieldWrapper (in IKVM.Internal)
      CompiledPropertyFieldWrapper (in IKVM.Internal)
      ConstantFieldWrapper (in IKVM.Internal)
      DynamicPropertyFieldWrapper (in IKVM.Internal)
      EnumFieldWrapper (in IKVM.Internal.DotNetTypeWrapper.EnumEnumTy
      EnumValueFieldWrapper (in IKVM.Internal.DotNetTypeWrapper)
      GetterFieldWrapper (in IKVM.Internal)
      SimpleFieldWrapper (in IKVM.Internal)
      VolatileLongDoubleFieldWrapper (in IKVM.Internal)
    MethodWrapper (in IKVM.Internal)
```

Figure 6: Subclasses of `FieldWrapper`, Sec. 1.3.2



```
Object (in System)
  MemberWrapper (in IKVM.Internal)
    FieldWrapper (in IKVM.Internal)
    MethodWrapper (in IKVM.Internal)
      BaseFinalMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      CloneMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      ConstructorForwarder (in IKVM.Internal.AotTypeWrapper)
      DelegateConstructorMethodWrapper (in IKVM.Internal.DynamicTypeWrapper.JavaTypeImpl)
      DelegateConstructorMethodWrapper (in IKVM.Internal.CompiledTypeWrapper)
      DelegateMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      DummyMethodWrapper (in IKVM.Internal)
      DynamicMethodWrapper (in Compiler)
      DynamicOnlyMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
        AttributeAnnotationMethodWrapper (in IKVM.Internal.DotNetTypeWrapper.AttributeAnno
      EnumValueOfMethodWrapper (in IKVM.Internal.DotNetTypeWrapper.EnumEnumTypeWrapper
      EnumValuesMethodWrapper (in IKVM.Internal.DotNetTypeWrapper.EnumEnumTypeWrapper)
      EnumWrapMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      FinalizeMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      MulticastDelegateCtorMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
      RemappedMethodBaseWrapper (in IKVM.Internal.CompilerClassLoader.RemapperTypeWrappe
        RemappedConstructorWrapper (in IKVM.Internal.CompilerClassLoader.RemapperTypeWra
        RemappedMethodWrapper (in IKVM.Internal.CompilerClassLoader.RemapperTypeWrapper
      ReplacedMethodWrapper (in IKVM.Internal.AotTypeWrapper)
      SimpleCallMethodWrapper (in IKVM.Internal)
      SmartMethodWrapper (in IKVM.Internal)
        ByRefMethodWrapper (in IKVM.Internal.DotNetTypeWrapper)
        CompiledRemappedMethodWrapper (in IKVM.Internal.CompiledTypeWrapper)
        GhostMethodWrapper (in IKVM.Internal.MethodWrapper)
        SmartCallMethodWrapper (in IKVM.Internal)
        SmartConstructorMethodWrapper (in IKVM.Internal)
      ValueTypeDefaultCtor (in IKVM.Internal.DotNetTypeWrapper)
```

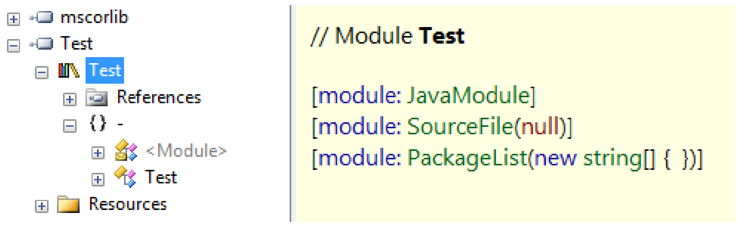
Figure 7: Subclasses of `MethodWrapper`, Sec. 1.3.2



```
[STAThread]
public static int main()
{
    int num;
    try
    {
        Startup.enterMainThread();
        Test.main(Startup.glob());
    }
    catch (Exception exception1)
    {
        Exception e = Util.mapException(exception1);
        Thread t = Thread.currentThread();
        t.getThreadGroup().uncaughtException(t, e);
        num = 1;
    }
    finally
    {
        Startup.exitMainThread();
    }
    return num;
}
```

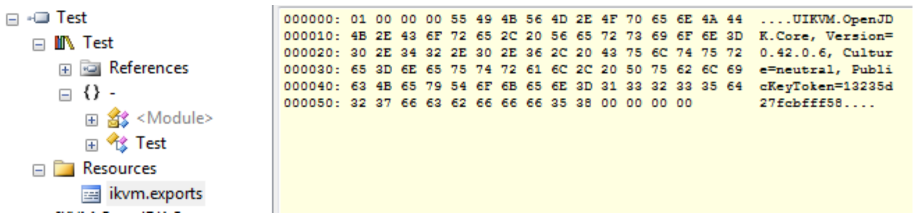
Figure 8: What `mb.CreateGlobalFunctions()` does, Sec. 1.3.3

- lines 526 to 543: add a `class.map` resource or (more commonly) add a `JavaModule` attribute.
- lines 545 to 558, add a package list and export map. Together with the attribute from the previous step, looks as follows:



```
// Module Test

[module: JavaModule]
[module: SourceFile(null)]
[module: PackageList(new string[] { })]
```

Additionally, the `ikvm.exports` resource is emitted (at the end of `CompilerClassLoader.WriteExportMap()`



```
000000: 01 00 00 00 55 49 4B 56 4D 2E 4F 70 65 6E 4A 44  ....UIKVM.OpenJD
000010: 4B 2E 43 6F 72 65 2C 20 56 65 72 73 69 6F 6E 3D  K.Core, Version=
000020: 30 2E 34 32 2E 30 2E 36 2C 20 43 75 6C 74 75 72  0.42.0.6, Cultur
000030: 65 3D 6E 65 75 74 72 61 6C 2C 20 50 75 62 6C 69  e=neutral, Publi
000040: 63 4B 65 79 54 6F 6B 65 6E 3D 31 33 32 33 35 64  cKeyToken=13235d
000050: 32 37 66 63 62 66 66 66 35 38 00 00 00 00        27fcbfff58....
```

- lines 560 to 571, serialization of binary assembly, no transformation performed.

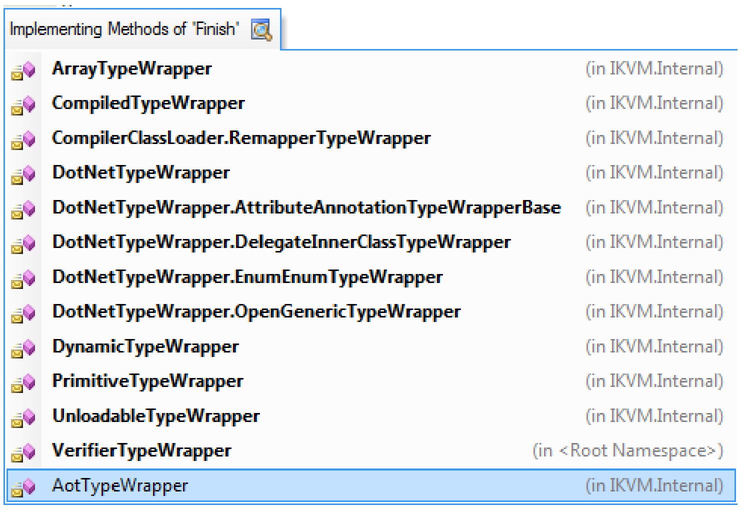
Listing 3: Where `TypeWrapper.Finish()` is invoked, Sec. 2

```
// C# code, from CompilerClassLoader.Compile()
List<TypeWrapper> allwrappers = new List<TypeWrapper>();
foreach(string s in classesToCompile)
{
   TypeWrapper wrapper = LoadClassByDottedNameFast(s);
   if(wrapper != null)
   {
      ClassLoaderWrapper loader = wrapper.GetClassLoader();
      if(loader != this) { . . . }
      if(map == null)
      {
         wrapper.Finish(); /*- code generation goes on here!
                              Line 2929 in CompilerClassLoader.cs */
      }
      int pos = wrapper.Name.LastIndexOf('.');
      . . .
      allwrappers.Add(wrapper);
   }
}
```

2 Before IL code generation

We had postponed in Sec. 1.2.2 the discussion about IL codegen. Listing 3 shows where `TypeWrapper.Finish()` is invoked during `CompilerClassLoader.Compile()` (actually that method is abstract, so one of its overrides is run).

Although we focus first on IL generation as resulting from the override `AotTypeWrapper.Finish()`, in fact there a number of such overrides, and in principle all are necessary to understand how `ikvmc` works:


Implementing Methods of 'Finish'

ArrayTypeWrapper	(in IKVM.Internal)
CompiledTypeWrapper	(in IKVM.Internal)
CompilerClassLoader.RemapperTypeWrapper	(in IKVM.Internal)
DotNetTypeWrapper	(in IKVM.Internal)
DotNetTypeWrapper.AttributeAnnotationTypeWrapperBase	(in IKVM.Internal)
DotNetTypeWrapper.DelegateInnerClassTypeWrapper	(in IKVM.Internal)
DotNetTypeWrapper.EnumEnumTypeWrapper	(in IKVM.Internal)
DotNetTypeWrapper.OpenGenericTypeWrapper	(in IKVM.Internal)
DynamicTypeWrapper	(in IKVM.Internal)
PrimitiveTypeWrapper	(in IKVM.Internal)
UnloadableTypeWrapper	(in IKVM.Internal)
VerifierTypeWrapper	(in <Root Namespace>)
AotTypeWrapper	(in IKVM.Internal)

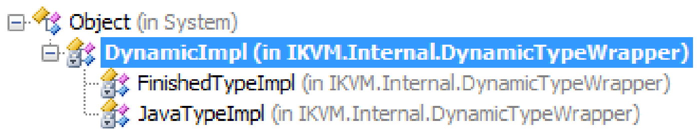
Rather than covering all things codegen, we checkpoint what goes on at a few program points, devoting a subsection to each.

Listing 4: What an JavaTypeImpl can access, Sec. 2.1

```
private sealed class JavaTypeImpl : DynamicImpl
{
   private readonly ClassFile classFile;
   private readonly DynamicTypeWrapper wrapper;
   private TypeBuilder typeBuilder;
   private MethodWrapper[] methods;
   private MethodWrapper[] baseMethods;
   private FieldWrapper[] fields;

   private FinishedTypeImpl finishedType; /*- near the end of JavaTypeImpl.FinishCore(),
                                               this field is assigned a FinishedTypeImpl */

   private bool finishInProgress;
   private Dictionary<string, string> memberclashtable;
   private MethodBuilder clinitMethod;
   private MethodBuilder finalizeMethod;
    . . .
```

2.1 Setting up the scene: DynamicTypeWrapper constructor

We will not discuss in detail how the scene for codegen is set up (i.e., what goes on before TypeWrapper.Finish() is invoked during CompilerClassLoader.Compile()). Other than saying that as part of AotTypeWrapper's constructor a JavaTypeImpl is instantiated (and referenced from the impl field of that wrapper's DynamicTypeWrapper base class).


Object (in System)
DynamicImpl (in IKVM.Internal.DynamicTypeWrapper)
FinishedTypeImpl (in IKVM.Internal.DynamicTypeWrapper)
JavaTypeImpl (in IKVM.Internal.DynamicTypeWrapper)

We mention this because two of JavaTypeImpl's methods will show up in what follows (Finish() which invokes FinishCore(), this last method returning a FinishedTypeImpl). Listing 4 gives an idea about JavaTypeImpl.

2.2 Shortly after we have an AotTypeWrapper

The take home message from this subsection is: the fresh AotTypeWrapper is moved to new typestates by CreateStep1 and CreateStep2NoFail:

```
DynamicTypeWrapper type;
type = new AotTypeWrapper(f, (CompilerClassLoader)classLoader);
bool hasclinit;
type.CreateStep1(out hasclinit);
string mangledTypeName = AllocMangledName(f.Name);
// This step actually creates the TypeBuilder.
type.CreateStep2NoFail(hasclinit, mangledTypeName);
```

which as we can see below just delegate to the AotTypeWrapper's impl (that's one of the reasons why we had to single out JavaTypeImpl in the previous subsection):

```
internal void CreateStep1(out bool hasclinit)
{
   ((JavaTypeImpl)impl).CreateStep1(out hasclinit);
}

internal void CreateStep2NoFail(bool hasclinit, string mangledTypeName)
{
   ((JavaTypeImpl)impl).CreateStep2NoFail(hasclinit, mangledTypeName);
}
```

Summing up:

- during `CreateStep1`:
  - lines 427 to 493:
    * all classfile methods are visited, mapping method flags to IKVM's `MemberFlags` (which encode some IKVM-isms such as `MirandaMethod` (Sec. 7.3) and `CallerID`).
    * Methods are wrapped. If the type wrapper `IsGhost`, methods are wrapped in `GhostMethodWrappers` (other outcomes include `DelegateConstructorMethodWrapper` and `SmartConstructorMethodWrapper`), while the "usual case" is `SmartCallMethodWrapper`
  - line 494: it's determined whether `wrapper.HasStaticInitializer`
  - lines 495 to 527: Miranda methods are added (Sec. 7.3)
  - lines 528 to 559: field wrappers are created. Outcomes: `ConstantFieldWrapper`, `DynamicPropertyFieldWrapper`, `VolatileLongDoubleFieldWrapper`, `SimpleFieldWrapper` (but there are more kinds, Figure 6).
- during `CreateStep2NoFail`:
  - lines 568 to 580: based on the classfile attributes, what will be the type builder's attributes are determined
  - lines 590 to 654: something is done in case the classfile is an inner class (TODO: what?)
  - lines 655 to 679: for a public nested type, further type attributes are set depending on whether the outer type is public or not.
  - lines 686 to 746: with the information gathered so far, a `TypeBuilder` is instantiated and stored in the `JavaTypeImpl`'s `typeBuilder` field (both actual and declared type are `System.Reflection.Emit.TypeBuilder`, not IKVM's `Emit` counterpart).
  - lines 748 to 757: more things done for a nested type
  - lines 758 to 762: name change for a nested type (`$` vs. `+`)
  - lines 763 to 767: for a classfile defining an annotation, the type wrapper gets an `AnnotationBuilder` set.
  - lines 770 to 789: Quoting: "*For Java 5 Enum types, we generate a nested .NET enum. This is primarily to support annotations that take enum parameters.*"
  - lines 790 to 886: TODO

Quoting:

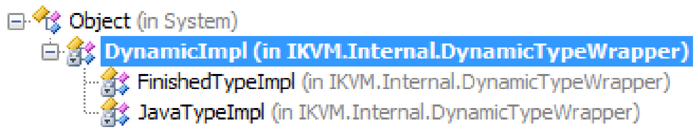
CreateStep2NoFail:

- *The inner class stuff is mostly related to robustness in the face of mismatching class files. So the inner/outer relationship is validated from both sides.*
- *`System.Reflection.Emit.TypeBuilder` is not actually used. All assembly generation in `ikvmc` is done with `IKVM.Reflection`, however, in dynamic mode (`IKVM.Runtime.dll`) the same code (mostly) is used with `System.Reflection.Emit`.*
- *Also, from an `ikvmc` point of view, `DynamicTypeWrapper` and `AotTypeWrapper` can be considered a single type. Most of the codegen is shared between ikvmc and `IKVM.Runtime` (that is represented by `DynamicTypeWrapper`), but some `ikvmc` specific things (like `map.xml` support) is in the `AotTypeWrapper` subclass.*

2.3 JavaTypeImpl's Finish() and FinishCore()

An execution path of interest starts with `AotTypeWrapper.Finish()` which in turn calls its super (after which it will check for non-nullness of `workaroundBaseClass` and if so `workaroundBaseClass.Finish()`)

The super-invocation means `DynamicTypeWrapper.Finish()` is run which amounts to `impl = impl.Finish()`, the topic of this subsection. Before returning, that method invokes `FinishCore()`, so in practice `Finish()` and `FinishCore()` always run in sequence. BTW, it must have to do with some restriction on narrowing down the return type, because `Finish()` is declared to return a `DynamicImpl` when in fact it will only return a `FinishedTypeImpl`. To recap:


Object (in System)
DynamicImpl (in IKVM.Internal.DynamicTypeWrapper)
FinishedTypeImpl (in IKVM.Internal.DynamicTypeWrapper)
JavaTypeImpl (in IKVM.Internal.DynamicTypeWrapper)

Back to `JavaTypeImpl.Finish()`:

- lines 1490 to 1493: grabs the type wrapper for the base class and invokes `Finish()` on it.
- lines 1495 to 1498: for a nested class, grabs the type wrapper for the outer class and invokes `Finish()` on it.
- lines 1510 to 1513: for each implemented interface, grabs its type wrapper and invokes `Finish()`
- lines 1514 to 1542: Quoting: "*make sure all classes are loaded, before we start finishing the type. During finishing, we may not run any Java code, because that might result in a request to finish the type that we are in the process of finishing, and this would be a problem.*"

After that the action moves to `JavaTypeImpl.FinishCore()`:

- lines 1554 to 1564: checking and returning from cyclic invocations
- line 1565: console output "`Finishing: <wrapper.Name>`"

- lines 1569 to 1610: obtain the type wrappers for contained inner classes (to be passed as constructor argument to `FinishedTypeImpl`)
- line 1611: a local `FinishContext` is created! (whose constructor just copies arguments to fields) Like this:

  ```
  FinishContext context = new FinishContext(classFile, wrapper, typeBuilder);
  ```

- lines 1613 to 1617: an `annotationBuilder` if present results in a `CustomAttributeBuilder` set in the `typeBuilder`
- lines 1618 to 1628: after the context is finished, the following block will be run (a block that is registered as a delegate instance with the context):

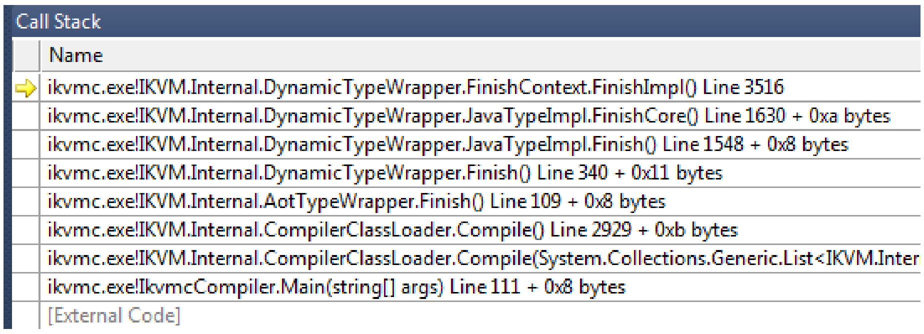
  ```
  if (enumBuilder != null) { enumBuilder.CreateType(); }
  if (annotationBuilder != null) { annotationBuilder.Finish(this); }
  ```

- the rest: More indirection! The action moves first to the context's `FinishImpl` (almost 600 LOC) and then to the constructor of `FinishedTypeImpl` (which, admittedly, just copies values from arguments to fields). This explains why the next section is devoted to `FinishContext.FinishImpl()`.

  ```
  Type type = context.FinishImpl();
  MethodInfo finishedClinitMethod = clinitMethod;
  finishedType = new FinishedTypeImpl(type, innerClassesTypeWrappers, declaringTypeWrapper,
                                      wrapper.ReflectiveModifiers,
                                      Metadata.Create(classFile),
                                      finishedClinitMethod, finalizeMethod,
                                      annotationBuilder, enumBuilder);
  return finishedType;
  ```

3 IL code generation

We're now at `FinishContext.FinishImpl()` (almost 600 LOC) whose completion will allow returning from `JavaTypeImpl`'s finishing methods, and from the type wrapper's finishing methods. More precisely:


Call Stack

```
Name
ikvmc.exe!IKVM.Internal.DynamicTypeWrapper.FinishContext.FinishImpl() Line 3516
ikvmc.exe!IKVM.Internal.DynamicTypeWrapper.JavaTypeImpl.FinishCore() Line 1630 + 0xa bytes
ikvmc.exe!IKVM.Internal.DynamicTypeWrapper.JavaTypeImpl.Finish() Line 1548 + 0x8 bytes
ikvmc.exe!IKVM.Internal.DynamicTypeWrapper.Finish() Line 340 + 0x11 bytes
ikvmc.exe!IKVM.Internal.AotTypeWrapper.Finish() Line 109 + 0x8 bytes
ikvmc.exe!IKVM.Internal.CompilerClassLoader.Compile() Line 2929 + 0xb bytes
ikvmc.exe!IKVM.Internal.CompilerClassLoader.Compile(System.Collections.Generic.List<IKVM.Inter
ikvmc.exe!IkvmcCompiler.Main(string[] args) Line 111 + 0x8 bytes
[External Code]
```

This method stretches between lines 3515 and 4103.

- line 3520, TODO: `wrapper.FinishGhost(typeBuilder, methods)`. Sample run to show how methods for a ghost interface are rewritten.
- lines 3522 to 3588, "*if we're not abstract make sure we don't inherit any abstract methods*"
- lines 3589 to 3782: iterate over classfile methods (declared, not inherited, I guess). For each such method with index `i`:
  - lines 3597 to 3604: "*method doesn't really exist (e.g., delegate constructor or clinit that is optimized away)*"
  - lines 3605 to 3620: A constructor. Invoke `CompileConstructorBody(...)`. This method with 30 LOC checks whether `map.xml` is in effect (not our case) and if not `Compiler.Compile` (Sec. 3.1) is invoked to do the actual job of IL generation.
  - lines 3623 to 3644: Abstract method. In case (a) the declaring class is concrete or (b) the method is package accessible in a public class, consider it a stub and emit:

    ```
    ilGenerator.EmitThrow("java.lang.AbstractMethodError", classFile.Name + "." + m.Name + m.Signature);
    ```

  - lines 3646 to 3748: Native method. Use JNI.
  - lines 3751 to 3780: "Normal" method. Check whether `map.xml` is in effect (no) and invoke `Compiler.Compile` (Sec. 3.1)
- lines 3784 to 3812: `(clinitIndex != -1 || (basehasclinit  !classFile.IsInterface) || classFile.HasInitializedFields)`
- line 3815: "*add all interfaces that we implement (including the magic ones) and handle ghost conversions*" (details in Sec. 5.1). This is handled by invoking:

  ```
  ImplementInterfaces(wrapper.Interfaces, new List<TypeWrapper>());
  ```

- lines 3818 to 3887: `(!classFile.IsInterface)`. Details in Sec. 6. Basically:
  - lines 3820 to 3831: add a constructor if none there, because:

    ```
    // if a class has no constructor, we generate one otherwise Ref.Emit will create a default ctor
    // and that has several problems:
    // - base type may not have an accessible default constructor
    // - Ref.Emit uses BaseType.GetConstructors() which may trigger a TypeResolve event
    // - we don't want the synthesized constructor to show up in Java
    ```

  - lines 3833 to 3843: implement those interface methods that need implementation (Sec. 5.2)

    ```
    // here we loop thru all the interfaces to explicitly implement any methods that we inherit from
    // base types that may have a different name from the name in the interface
    // (e.g. interface that has an equals() method that should override System.Object.Equals())
    // also deals with interface methods that aren't implemented (generate a stub that throws AbstractMet
    // and with methods that aren't public (generate a stub that throws IllegalAccessError)
    ```

  - lines 3844 to 3854: complete incomplete interface implementations (also in Sec. 5.2):

```
// if any of our base classes has an incomplete interface implementation we need to look through all
// the base class interfaces to see if we've got an implementation now
```

  - lines 3855 to 3858: conditionally call `AddUnsupportedAbstractMethods()` (Sec. 6.1)
  - lines 3859 to 3885: for each method, check if adding an override is needed. Finally, `AddAutomagicSerialization(wrapper)` (Sec. 6.2)

- lines 3890 to 3929:

```
// If we're an interface that has public/protected fields, we create an inner class
// to expose these fields to C# (which stubbornly refuses to see fields in interfaces).
```

- lines 3931 to 3932: in `map.xml` mode (not our case), emit any additional metadata
- lines 3935 to 4021: for each method declared in the classfile:
  - add parameters, add annotations, and add parameter annotations.
- lines 4023 to 4052: for each field declared in the classfile, add annotation if any.
- lines 4054 to 4064: add class annotations if any.
- lines 4069 to 4092: `typeBuilder.CreateType()`, same for `typeCallerID` if any, run the delegates registered for "post finish processing". Additionally (lines 4087 to 4090) `if (classFile.IsInterface  !classFile.IsPublic)` then `DefineProxyHelper(type)`.
- line 4097: `wrapper.GetClassLoader().SetWrapperForType(type, wrapper)`
- line 4099: `wrapper.FinishGhostStep2()`
- line 4101: `BakedTypeCleanupHack.Process(wrapper)`

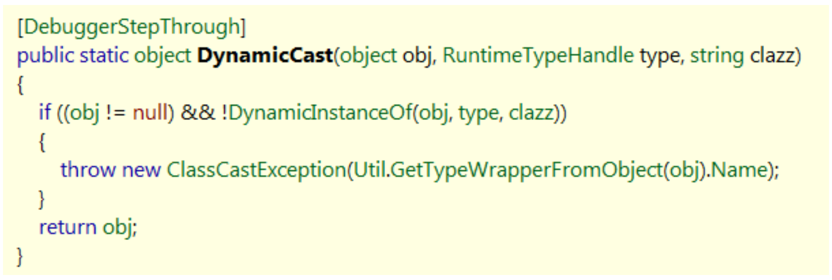
3.1 Static `Compiler.Compile(...)`, starting at line 818 in `compiler.cs`

This method stretches between lines 818 and 972:

- lines 823 to 856: emitting debug info (line numbers)
- lines 857 to 869: for each formal parameter, grab the wrapper for its type and if `IsUnloadable`

```
ilGenerator.Emit(OpCodes.Ldarg, (short)(i + (m.IsStatic ? 0 : 1)));
ilGenerator.Emit(OpCodes.Ldtoken, clazz.TypeAsTBD);
ilGenerator.Emit(OpCodes.Ldstr, args[i].Name);
ilGenerator.Emit(OpCodes.Call, ByteCodeHelperMethods.DynamicCast); // IKVM.Runtime.ByteCodeHelper.DynamicC
ilGenerator.Emit(OpCodes.Pop);
```

in turn, `DynamicCast(...)` is defined as follows in `IKVM.Runtime.dll`:



```
[DebuggerStepThrough]
public static object DynamicCast(object obj, RuntimeTypeHandle type, string clazz)
{
   if ((obj != null) && !DynamicInstanceOf(obj, type, clazz))
   {
      throw new ClassCastException(Util.GetTypeWrapperFromObject(obj).Name);
   }
   return obj;
}
```

- lines 870 to 904: Oh well. We'll just have to dig deeper to find out about IL generation. A `Compiler` is instantiated and its constructor does a lot of work (Sec. 3.1.1)
- lines 906 to 967:
  - for a synchronized static method, bracket the IL for its body with monitor enter and exit,
  - otherwise just emit IL for the method body (a `Block`). BTW, `Block.Leave` emits sthg non-trivial.
  - in both cases above, the *long awaited* instruction-level codegen takes place! How? Take a look at Sec. 3.2.
  - update line-number table.

3.1.1 Compiler constructor, lines 264 to 610 in `compiler.cs`

- line 286, no-op because we're not in `map.xml` mode: (TODO confirm) `replacedMethodWrappers = clazz.GetReplacedMethodsFor(mw)`
- line 296, `ma = new MethodAnalyzer(clazz, mw, classFile, m, classLoader);` that's a whooping 1620 LOC, but no IL generation takes place there. Details in Sec. 3.1.2.
- lines 306 to 338, for each local var let `tw` be the type wrapper for its declared type:

```
if(!tw.IsUnloadable &&
   v.type != VerifierTypeWrapper.UninitializedThis &&
   (v.type != tw || tw.TypeAsLocalOrStackType != tw.TypeAsSignatureType))
{
   v.builder = ilGenerator.DeclareLocal(v.type.TypeAsLocalOrStackType);
   if(debug && v.name != null)
   {
      v.builder.SetLocalSymInfo(v.name);
   }
   v.isArg = false;
   ilGenerator.Emit(OpCodes.Ldarg_S, (byte)arg);
   tw.EmitConvSignatureTypeToStackType(ilGenerator);
   ilGenerator.Emit(OpCodes.Stloc, v.builder);
}
```

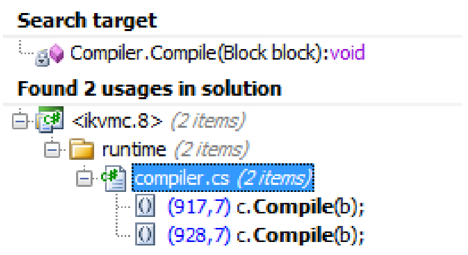
- lines 340 to 610, conversion of exception table entries.

3.1.2 Method Analyzer, lines 1219 to 2686 in `verifier.cs`

- lines 1219 to 2290: compute type stacks for the method received as argument
- lines 2292 to 2327: Optimization pass to remove `asserts`
- lines 2328 to 2566: Compute reachability at the instruction level
- lines 2568 to 2685: Quoting: "*now that we've done the code flow analysis, we can do a liveness analysis on the local variables*"

3.2 The long awaited instruction-level compilation

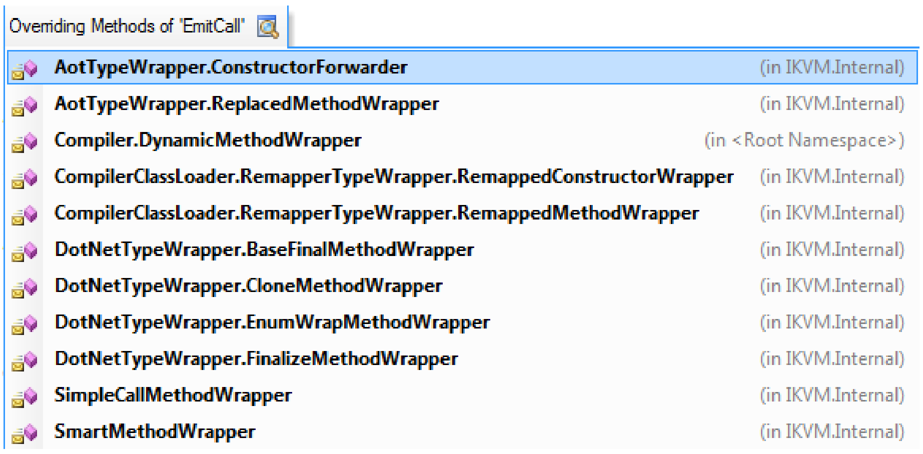
The instance method `Compiler.Compile(Compiler.Block block)` is in charge of that. It stretches between lines 1210 to 3054 in `compiler.cs`. This method is invoked from just two places, both inside the method described in Sec. 3.1:



```
Search target
  Compiler.Compile(Block block):void
Found 2 usages in solution
  <ikvmc.8> (2 items)
    runtime (2 items)
      compiler.cs (2 items)
        (917,7) c.Compile(b);
        (928,7) c.Compile(b);
```

The method's instructions are iterated (not just those in the `block` received as argument).

- lines 1220 to 1230: Auxiliary arrays indicate how many scopes begin and end at the current instruction.
- lines 1234 to 1373: instructions are emitted to effect leaving synchronization blocks and exception handlers.
- lines 1392 to 1410: more CIL-relevant code: "*if the instruction is only backward reachable, ECMA says it must have an empty stack, so we move the stack to locals*"
- lines 1417 to 1441: "*if we're entering an exception block, we need to setup the exception block and transfer the stack into it*"
- lines 1443 to 1466: line numbers
- lines 1468 to 1521: handle the following case: "*JSR 133 specifies that a finalizer cannot run while the constructor is still in progress.*"
- lines 1524 to 3023: the main instruction switch. The transformations falling under this item are covered in Sec. 4.
- lines 3026 to 3052: codegen artifact: "*mark next instruction as inuse*"



```
Overriding Methods of 'EmitCall'
AotTypeWrapper.ConstructorForwarder                                   (in IKVM.Internal)
AotTypeWrapper.ReplacedMethodWrapper                                  (in IKVM.Internal)
Compiler.DynamicMethodWrapper                                         (in <Root Namespace>)
CompilerClassLoader.RemapperTypeWrapper.RemappedConstructorWrapper    (in IKVM.Internal)
CompilerClassLoader.RemapperTypeWrapper.RemappedMethodWrapper         (in IKVM.Internal)
DotNetTypeWrapper.BaseFinalMethodWrapper                              (in IKVM.Internal)
DotNetTypeWrapper.CloneMethodWrapper                                  (in IKVM.Internal)
DotNetTypeWrapper.EnumWrapMethodWrapper                               (in IKVM.Internal)
DotNetTypeWrapper.FinalizeMethodWrapper                               (in IKVM.Internal)
SimpleCallMethodWrapper                                               (in IKVM.Internal)
SmartMethodWrapper                                                    (in IKVM.Internal)
```

Figure 9: Overrides for `MethodWrapper::EmitCall(ILGenerator)`

4 Transformations during instruction-level compilation

Under item "`lines 1524 to 3023: the main instruction switch`" in Sec. 3.2 are hidden a number of rewritings.

4.1 Callsite rewriting

We first look in this subsection at rewritings related to callsites. There are several such rewritings, each triggered by a different pattern. What they share in common is that they are all to be found somewhere in the handler (lines 1726 to 2011 in `compiler.cs`) for the following case:

```
case NormalizedByteCode.__dynamic_invokeinterface:
case NormalizedByteCode.__dynamic_invokevirtual:
case NormalizedByteCode.__dynamic_invokespecial:
case NormalizedByteCode.__invokevirtual:
case NormalizedByteCode.__invokeinterface:
case NormalizedByteCode.__invokespecial:
```

For some of these rewritings we devote a sub-subsection, for the rest an item in the following list:

- lines 1760 to 1790 (in `compiler.cs`). `castclass` opcodes are inserted to downcast an actual argument to the interface of its formal parameter:

  ```
  // if the stack values don't match the argument types (for interface argument types)
  // we must emit code to cast the stack value to the interface type
  ```

- lines 1792 to 2009: `invokespecial` of an instance constructor. To know what code gets emitted by `method.EmitCall(ilGenerator)`, knowing the actual override is necessary (Figure 9).

4.1.1 Redirect invocations on protected methods from `java.lang.Object` or `Throwable` to its .NET base type

We're at lines 1740 to 1757 in `compiler.cs`:

```
if(method.IsProtected && (     method.DeclaringType == java_lang_Object
                           || method.DeclaringType == java_lang_Throwable))
{
   // HACK we may need to redirect finalize or clone from java.lang.Object/Throwable
   // to a more specific base type.
   if(thisType.IsAssignableTo(cli_System_Object))
   {
      method = cli_System_Object.GetMethodWrapper(cpi.Name, cpi.Signature, true);
   }
   else if(thisType.IsAssignableTo(cli_System_Exception))
   {
      method = cli_System_Exception.GetMethodWrapper(cpi.Name, cpi.Signature, true);
   }
   else if(thisType.IsAssignableTo(java_lang_Throwable))
   {
      method = java_lang_Throwable.GetMethodWrapper(cpi.Name, cpi.Signature, true);
   }
}
```

One might get the feeling the above doesn't tell the whole story because `method` came from:

```
MethodWrapper method = GetMethodCallEmitter(cpi, instr.NormalizedOpCode);
```

which in principle might perform any conceivable redirection. But given that we're not in `map.xml` mode that *appears* not to be the case. Thus rewiring will be done after returning from `GetMethodCallEmitter(...)`, as done above (i.e., rewiring into `System.Object`, `System.Exception`, and `java.lang.Throwable`).

The comment in the snippet above refers to `finalize()` and `clone()` only because they are the only `protected` methods in `java.lang.Object`.

Quoting from line 3837 in `TypeWrapper.cs`: "*System.Object must appear to be derived from java.lang.Object*". On the other hand, if `type` is `java.lang.Object` its base type is `System.Object`. Contradiction? *Not at all!* I hope Sec. 7.1 can convince us on this.

4.2 Object instantiation

We're at lines 2172 to 2191 in `compiler.cs`:

```
case NormalizedByteCode.__new:
{
   TypeWrapper wrapper = classFile.GetConstantPoolClassType(instr.Arg1);
   if(wrapper.IsUnloadable)
   {
      Profiler.Count("EmitDynamicNewCheckOnly");
      // this is here to make sure we throw the exception in the right location (before
      // evaluating the constructor arguments)
      ilGenerator.Emit(OpCodes.Ldtoken, clazz.TypeAsTBD);
      ilGenerator.Emit(OpCodes.Ldstr, wrapper.Name);
      ilGenerator.Emit(OpCodes.Call, ByteCodeHelperMethods.DynamicNewCheckOnly);
   }
   else if(wrapper != clazz)
   {
      // trigger cctor (as the spec requires)
      wrapper.EmitRunClassConstructor(ilGenerator);
   }
   // we don't actually allocate the object here, the call to <init> will be converted
   // into a newobj instruction
   break;
}
```

4.3 TODO: Array[Object] creation, in particular for ghost element type

Lines 2237 – 2273 in `compiler.cs`, `(wrapper.IsGhost || wrapper.IsGhostArray)`

```
case NormalizedByteCode.__multianewarray:
{
   LocalBuilder localArray = ilGenerator.UnsafeAllocTempLocal(typeof(int[]));
   LocalBuilder localInt = ilGenerator.UnsafeAllocTempLocal(typeof(int));
   ilGenerator.LazyEmitLdc_I4(instr.Arg2);
   ilGenerator.Emit(OpCodes.Newarr, typeof(int));
   ilGenerator.Emit(OpCodes.Stloc, localArray);
   for(int j = 1; j <= instr.Arg2; j++)
   {
      ilGenerator.Emit(OpCodes.Stloc, localInt);
      ilGenerator.Emit(OpCodes.Ldloc, localArray);
      ilGenerator.LazyEmitLdc_I4(instr.Arg2 - j);
      ilGenerator.Emit(OpCodes.Ldloc, localInt);
      ilGenerator.Emit(OpCodes.Stelem_I4);
   }
   TypeWrapper wrapper = classFile.GetConstantPoolClassType(instr.Arg1);
   if(wrapper.IsUnloadable)
   {
      Profiler.Count("EmitDynamicMultianewarray");
      ilGenerator.Emit(OpCodes.Ldtoken, clazz.TypeAsTBD);
      ilGenerator.Emit(OpCodes.Ldstr, wrapper.Name);
      ilGenerator.Emit(OpCodes.Ldloc, localArray);
      ilGenerator.Emit(OpCodes.Call, ByteCodeHelperMethods.DynamicMultianewarray);
   }
   else if(wrapper.IsGhost || wrapper.IsGhostArray)
   {
      TypeWrapper tw = wrapper;
      while(tw.IsArray)
      {
         tw = tw.ElementTypeWrapper;
      }
      ilGenerator.Emit(OpCodes.Ldtoken, ArrayTypeWrapper.MakeArrayType(tw.TypeAsTBD, wrapper.ArrayRank));
      ilGenerator.Emit(OpCodes.Ldloc, localArray);
      ilGenerator.Emit(OpCodes.Call, ByteCodeHelperMethods.multianewarray_ghost);
      ilGenerator.Emit(OpCodes.Castclass, wrapper.TypeAsArrayType);
   }
   else
   {
      Type type = wrapper.TypeAsArrayType;
      ilGenerator.Emit(OpCodes.Ldtoken, type);
      ilGenerator.Emit(OpCodes.Ldloc, localArray);
      ilGenerator.Emit(OpCodes.Call, ByteCodeHelperMethods.multianewarray);
      ilGenerator.Emit(OpCodes.Castclass, type);
   }
   break;
}
```

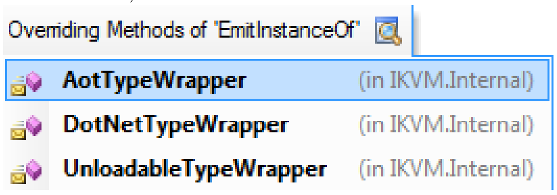
4.4 `instanceof`

The translation of check-cast by `ikvmc` shows once more that the actual code emitted depends on the runtime wrappers used.

Lines 2312 to 2317 in `compiler.cs` read:

```
case NormalizedByteCode.__instanceof:
{
   TypeWrapper wrapper = classFile.GetConstantPoolClassType(instr.Arg1);
   wrapper.EmitInstanceOf(clazz, ilGenerator);
   break;
}
```

Problem is, there are three overrides of `EmitInstanceOf`:



Overriding Methods of 'EmitInstanceOf'

AotTypeWrapper (in IKVM.Internal)
DotNetTypeWrapper (in IKVM.Internal)
UnloadableTypeWrapper (in IKVM.Internal)

4.5 Nitpick: Taking the remainder of dividing by -1

Lines 2586 to 2613 in `compiler.cs`:

```
case NormalizedByteCode.__lrem:
{
   // we need to special case taking the remainder of dividing by -1,
   // because the CLR rem instruction throws an OverflowException when
   // taking the remainder of dividing Int32.MinValue by -1, and
   // Java just silently overflows
   ilGenerator.Emit(OpCodes.Dup);
   ilGenerator.Emit(OpCodes.Ldc_I4_M1);
   if(instr.NormalizedOpCode == NormalizedByteCode.__lrem)
   {
      ilGenerator.Emit(OpCodes.Conv_I8);
   }
   CodeEmitterLabel label = ilGenerator.DefineLabel();
   ilGenerator.Emit(OpCodes.Bne_Un_S, label);
   ilGenerator.Emit(OpCodes.Pop);
   ilGenerator.Emit(OpCodes.Pop);
   ilGenerator.Emit(OpCodes.Ldc_I4_0);
   if(instr.NormalizedOpCode == NormalizedByteCode.__lrem)
   {
      ilGenerator.Emit(OpCodes.Conv_I8);
   }
   CodeEmitterLabel label2 = ilGenerator.DefineLabel();
   ilGenerator.Emit(OpCodes.Br_S, label2);
   ilGenerator.MarkLabel(label);
   ilGenerator.Emit(OpCodes.Rem);
   ilGenerator.MarkLabel(label2);
   break;
}
```

4.6 TODO: Throwing exceptions

Lines 2882 to 2896 in `compiler.cs`:

```
case NormalizedByteCode.__athrow_no_unmap:
   if(ma.GetRawStackTypeWrapper(i, 0).IsUnloadable)
   {
      ilGenerator.Emit(OpCodes.Castclass, typeof(Exception));
   }
   ilGenerator.Emit(OpCodes.Throw);
   break;
case NormalizedByteCode.__athrow:
   if(ma.GetRawStackTypeWrapper(i, 0).IsUnloadable)
   {
      ilGenerator.Emit(OpCodes.Castclass, typeof(Exception));
   }
   ilGenerator.Emit(OpCodes.Call, unmapExceptionMethod);
   ilGenerator.Emit(OpCodes.Throw);
   break;
```

For the record, `__athrow_no_unmap` is a *pseudo-bytecode*, of which there are others, as defined in the enum `NormalizedByteCode` in `ByteCode.cs`:

```
// This is where the pseudo-bytecodes start
__intrinsic_gettypehandlevalue = 239,
__fastore_conv = 240,
__fstore_conv = 241,
__dastore_conv = 242,
__dstore_conv = 243,
__athrow_no_unmap = 244,
__dynamic_getstatic = 245,
__dynamic_putstatic = 246,
__dynamic_getfield = 247,
__dynamic_putfield = 248,
__dynamic_invokeinterface = 249,
__dynamic_invokestatic = 250,
__dynamic_invokevirtual = 251,
__dynamic_invokespecial = 252,
__clone_array = 253,
__static_error = 254, // not a real instruction,
                      // this signals an instruction that is compiled as an exception
__iconst = 255
```

5 Magic around interfaces

5.1 "Magic" JDK interfaces that imply a .NET interface (1 of 2)

Method `ImplementInterfaces()` stretches between lines 4733 and 4806 in `DynamicTypeWrapper.cs` and is invoked from line 3815 in `FinishContext.FinishImpl()` (Sec. 3), preceded by the source comment: "*add all interfaces that we implement (including the magic ones) and handle ghost conversions*".

- line 4742,
  `if (!iface.IsPublic  !ReflectUtil.IsSameAssembly(ifaceType, typeBuilder)).`
  TODO: what are the `"__<Proxy>+"` interfaces? (Sec. 5.1.1)

- line 4746, `AddInterfaceImplementation` adds an entry to the assembly table (Metadata, Partition II) that tracks such low-level stuff.
- if the current (Java non-interface) type implements ... but not ...:
  - `java.lang.Iterable`, `System.Collections.IEnumerable` (Sec. 5.1.2)
  - `java.io.Closeable`, `System.IDisposable` (Sec. 5.1.3)
- "*if we implement a ghost interface, add an implicit conversion to the ghost reference value type*". About ghosts in general see Sec. 5.1.5, about this transformation see Sec. 5.1.4.
- do the above recursively for the base interfaces of each implemented interface.

5.1.1 TODO: `__<Proxy>+` interfaces

TODO: Not mentioned at all in `map.xml`

5.1.2 `java.lang.Iterable`

Codegen code shown in Listing 5 on p. 29. Example:

```
public class Test implements java.lang.Iterable {
  @Override public Iterator iterator() { return null; }
}
```

The output type has:

- `System.Collections.IEnumerable` (the non-generic one) added as implemented interface in addition to `java.lang.Iterable`
- an additional method with a dotted name (that happens to avoid collisions),

```
   System.Collections.IEnumerator
   'System.Collections.IEnumerable.GetEnumerator'
   ()
{
    return new ikvm.lang.IterableEnumerator(this);
}
```

- that method body will run when the implied method is called:

```
.override [mscorlib]System.Collections.IEnumerable::GetEnumerator
```

5.1.3 `java.io.Closeable`

Codegen code shown in Listing 6 on p. 30. Example:

```
public class Test implements java.io.Closeable {
  @Override public void close() throws IOException { }
}
```

The output type has:

Listing 5: Codegen for `java.lang.Iterable`, see Sec. 5.1.2

```
if (iface.Name == "java.lang.Iterable"
   && !wrapper.ImplementsInterface(ClassLoaderWrapper.GetWrapperFromType(typeof(System.Collections.IEnumerable))
{
  TypeWrapper enumeratorType = ClassLoaderWrapper
                               .GetBootstrapClassLoader()
                               .LoadClassByDottedNameFast("ikvm.lang.IterableEnumerator");
  if (enumeratorType != null)
  {
    typeBuilder.AddInterfaceImplementation(typeof(System.Collections.IEnumerable));
/*- add to metadata table */
    // FXBUG we're using the same method name as the C# compiler here because
    // both the .NET and Mono implementations of Xml serialization depend on this method name
    MethodBuilder mb =
           typeBuilder.DefineMethod("System.Collections.IEnumerable.GetEnumerator",
                                        MethodAttributes.Private | MethodAttributes.Virtual |
                                               MethodAttributes.NewSlot | MethodAttributes.Final | MethodAttribu
                                        typeof(System.Collections.IEnumerator),
                                        Type.EmptyTypes);
    AttributeHelper.HideFromJava(mb);
    typeBuilder.DefineMethodOverride(mb,
                                        typeof(System.Collections.IEnumerable).GetMethod("GetEnumerator"));
    CodeEmitter ilgen = CodeEmitter.Create(mb);
    ilgen.Emit(OpCodes.Ldarg_0);
    MethodWrapper mw = enumeratorType.GetMethodWrapper("<init>",
                                                      "(Ljava.lang.Iterable;)V",
                                                      false);
    mw.Link();
    mw.EmitNewobj(ilgen);
    ilgen.Emit(OpCodes.Ret);
  }
}
```

- `System.IDisposable` added as implemented interface in addition to `java.io.Closeable`
- an additional method with a fixed name that anyway avoids collisions,

  ```
  private void __<>Dispose() { this.close(); }
  ```

- that method body will run when the implied method is called:

  ```
   .override [mscorlib]System.IDisposable::Dispose
  ```

5.1.4 Adding implicit conversion due to ghost interface

We'll add implicits to `Predef`, instead of generating the conversion method as `ikvmc` does.

Lines 4786 to 4798 in `DynamicTypeWrapper.cs`:

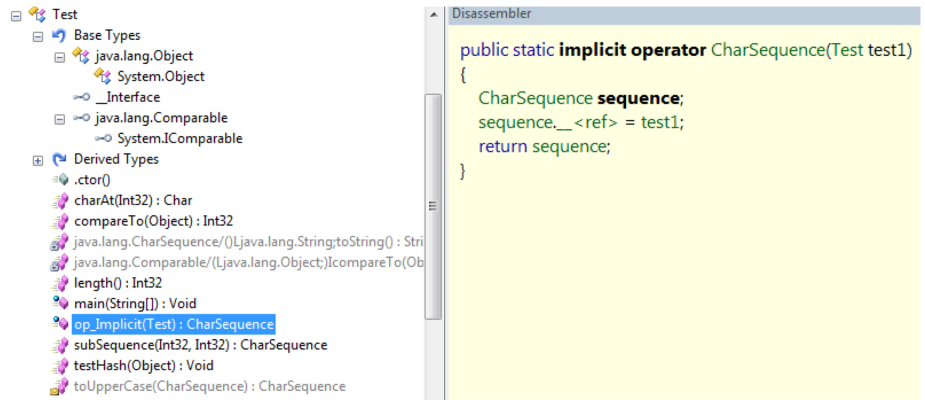
```
// if we implement a ghost interface, add an implicit conversion to the ghost reference value type
if (iface.IsGhost && wrapper.IsPublic)
{
  MethodBuilder mb = typeBuilder.DefineMethod("op_Implicit",
                                            MethodAttributes.HideBySig | MethodAttributes.Public |
                                                 MethodAttributes.Static |MethodAttributes.SpecialName,
```

Listing 6: Codegen for `java.io.Closeable`, see Sec. 5.1.3

```
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.SpecialName,
                                  typeof(void),
                                 Type.EmptyTypes);
  typeBuilder.DefineMethodOverride(mb,
                                  typeof(IDisposable).GetMethod("Dispose"));
  CodeEmitter ilgen = CodeEmitter.Create(mb);
  ilgen.Emit(OpCodes.Ldarg_0);
  MethodWrapper mw = iface.GetMethodWrapper("close",
                                          "()V",
                                          false);
  mw.Link();
  mw.EmitCallvirt(ilgen);
  ilgen.Emit(OpCodes.Ret);
}
```

```
                                          iface.TypeAsSignatureType,
                                          new Type[] { wrapper.TypeAsSignatureType });
  CodeEmitter ilgen = CodeEmitter.Create(mb);
  LocalBuilder local = ilgen.DeclareLocal(iface.TypeAsSignatureType);
  ilgen.Emit(OpCodes.Ldloca, local);
  ilgen.Emit(OpCodes.Ldarg_0);
  ilgen.Emit(OpCodes.Stfld, iface.GhostRefField);
  ilgen.Emit(OpCodes.Ldloca, local);
  ilgen.Emit(OpCodes.Ldobj, iface.TypeAsSignatureType);
  ilgen.Emit(OpCodes.Ret);
}
```

Assuming we had "`public class Test implements CharSequence ...`" the above results in:



```
Test
  Base Types
    java.lang.Object
      System.Object
    __Interface
    java.lang.Comparable
      System.IComparable
  Derived Types
  .ctor()
  charAt(Int32) : Char
  compareTo(Object) : Int32
  java.lang.CharSequence/()Ljava.lang.String;toString() : Stri
  java.lang.Comparable/(Ljava.lang.Object;)IcompareTo(Ob
  length() : Int32
  main(String[]) : Void
  op_Implicit(Test) : CharSequence
  subSequence(Int32, Int32) : CharSequence
  testHash(Object) : Void
  toUpperCase(CharSequence) : CharSequence
```

Disassembler

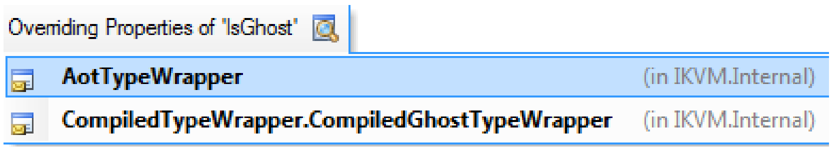
```
public static implicit operator CharSequence(Test test1)
{
    CharSequence sequence;
    sequence.__<ref> = test1;
    return sequence;
}
```

5.1.5 Generalities about ghost interfaces

Comment for `TypeWrapper.IsGhost`:

> *a ghost is an interface that appears to be implemented by a .NET type (e.g. `System.String` appears to implement JDK's `java.lang.CharSequence`, so `java.lang.CharSequence` is a ghost)*

Just two overrides:



Overriding Properties of 'IsGhost'

AotTypeWrapper	(in IKVM.Internal)
CompiledTypeWrapper.CompiledGhostTypeWrapper	(in IKVM.Internal)

A one-to-many map (`ghosts` in `CompilerClassLoader`) tracks for a ghost name which JDK classes implement it. Details in Listing 7 on p. 40 (although the methods listed run only in `map.xml` mode, they still convey the idea). Given that every non-null Java array is serializable and cloneable (details below), `AddGhost` is also called for the erased array type with those two ghosts (again, Listing 7).

```
class ArrayIsSerializable
{
  public static void main( String args[] )
  {
    int f1[] = null;
    int f2[] = new int[10];

    Serializable s = (Serializable)f1;
    System.out.println( s ); // null

    boolean b1 = f1 instanceof Serializable;
    boolean b2 = f2 instanceof Serializable;
    System.out.println( b1 ); // false
    System.out.println( b2 ); // true
  }
}
```

5.2 Generate overrides for methods of implied interfaces (2 of 2)

`ImplementInterfaceMethodStubs(...)` is invoked twice between lines 3842 and 3851 in `FinishContext.FinishImpl()` (Sec. 3):

```
// here we loop thru all the interfaces to explicitly implement any methods that we inherit from
// base types that may have a different name from the name in the interface
// (e.g. interface that has an equals() method that should override System.Object.Equals())
// also deals with interface methods that aren't implemented (generate a stub that throws AbstractMethodError)
// and with methods that aren't public (generate a stub that throws IllegalAccessError)

Dictionary<TypeWrapper, TypeWrapper>
  doneSet = new Dictionary<TypeWrapper, TypeWrapper>();

TypeWrapper[] interfaces = wrapper.Interfaces;

for (int i = 0; i < interfaces.Length; i++)
{
  ImplementInterfaceMethodStubs(doneSet, interfaces[i]); /*- first time */
}

// if any of our base classes has an incomplete interface implementation we need to look through all
// the base class interfaces to see if we've got an implementation now
TypeWrapper baseTypeWrapper = wrapper.BaseTypeWrapper;
while (baseTypeWrapper.HasIncompleteInterfaceImplementation)
{
  for (int i = 0; i < baseTypeWrapper.Interfaces.Length; i++)
  {
    ImplementInterfaceMethodStubs(doneSet, baseTypeWrapper.Interfaces[i]); /*- second time */
  }
  baseTypeWrapper = baseTypeWrapper.BaseTypeWrapper;
}
```

`ImplementInterfaceMethodStubs(...)` stretches between lines 4105 and 4127 in `DynamicTypeWrapper.cs`. It delegates to `ImplementInterfaceMethodStubImpl(method)` (Sec. 5.2.1).

```
private void ImplementInterfaceMethodStubs(Dictionary<TypeWrapper, TypeWrapper> doneSet,
                                                      TypeWrapper interfaceTypeWrapper)
{
  Debug.Assert(interfaceTypeWrapper.IsInterface);
  // make sure we don't do the same method twice
  if (doneSet.ContainsKey(interfaceTypeWrapper)) { return; }
  doneSet.Add(interfaceTypeWrapper, interfaceTypeWrapper);

  foreach (MethodWrapper method in interfaceTypeWrapper.GetMethods())
  {
    if (!method.IsStatic && !method.IsDynamicOnly)
    {
      ImplementInterfaceMethodStubImpl(method); /*- do the work */
    }
  }
  TypeWrapper[] interfaces = interfaceTypeWrapper.Interfaces;
  for (int i = 0; i < interfaces.Length; i++)
  {
    ImplementInterfaceMethodStubs(doneSet, interfaces[i]); /*- recurse over the current interface interfaces */
  }
}
```

5.2.1 TODO: `ImplementInterfaceMethodStubImpl(ifmethod)`

Now we have to do with a longer snippet. This method stretches between lines 4129 and 4243 in `DynamicTypeWrapper.cs`. What it does can be seen in Figure 10.

Let's say the `ifmethod` argument to `ImplementInterfaceMethodStubImpl(ifmethod)` is the wrapper for method `int Comparable.compareTo(Object)` (remember: IKVM = no generics).

- line 4136: first of all a non-colliding method name is needed:
  `string mangledName = ifmethod.DeclaringType.Name + "/" + ifmethod.Signature + ifmethod.Name`
- lines 4137 to 4147: next, we have to find the Java method (user-provided) to be called when the equivalent .NET method is invoked. That Java method is identified by `mce` below:

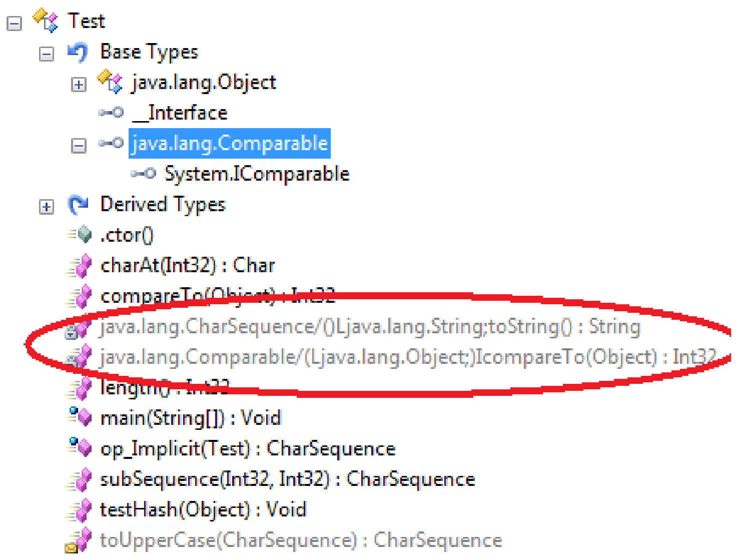

Test
Base Types
java.lang.Object
_Interface
java.lang.Comparable
System.IComparable
Derived Types
.ctor()
charAt(Int32) : Char
compareTo(Object) : Int32
java.lang.CharSequence/()Ljava.lang.String;toString() : String
java.lang.Comparable/(Ljava.lang.Object;)IcompareTo(Object) : Int32
length() : Int32
main(String[]) : Void
op_Implicit(Test) : CharSequence
subSequence(Int32, Int32) : CharSequence
testHash(Object) : Void
toUpperCase(CharSequence) : CharSequence

Figure 10: Methods added by `ImplementInterfaceMethodStubImpl(method)`, Sec. 5.2.1

```
MethodWrapper mce = null;
TypeWrapper lookup = wrapper;
while (lookup != null)
{
  mce = lookup.GetMethodWrapper(ifmethod.Name, ifmethod.Signature, true);
  if (mce == null || !mce.IsStatic)
  {
    break;
  }
  lookup = mce.DeclaringType.BaseTypeWrapper;
}
```

- lines 4180 to 4183: For Miranda methods see Sec. 7.3:

```
if (mce.IsMirandaMethod && mce.DeclaringType == wrapper)
{
  // Miranda methods already have a methodimpl (if needed) to implement the correct interface method
}
```

- lines 4196 to 4210: there are other cases (TODO), but this one is the most common:

```
else if (mce.GetMethod() == null || mce.RealName != ifmethod.RealName)
{
  MethodBuilder mb = typeBuilder.DefineMethod(mangledName,
                                              MethodAttributes.HideBySig | MethodAttributes.NewSlot | MethodAttr
                                                     MethodAttributes.Virtual | MethodAttributes.Final,
                                              ifmethod.ReturnTypeForDefineMethod,
                                              ifmethod.GetParametersForDefineMethod());
  AttributeHelper.HideFromJava(mb);
  CodeEmitter ilGenerator = CodeEmitter.Create(mb);
  ilGenerator.Emit(OpCodes.Ldarg_0);
  int argc = mce.GetParameters().Length;
  for (int n = 0; n < argc; n++)
  {
    ilGenerator.Emit(OpCodes.Ldarg_S, (byte)(n + 1));
  }
  mce.EmitCallvirt(ilGenerator);
  ilGenerator.Emit(OpCodes.Ret);
  typeBuilder.DefineMethodOverride(mb,
                                  (MethodInfo)ifmethod.GetMethod());
}
```

Example:

```
public class Test implements Comparable {
  @Override public int compareTo(Object arg0) { return 0; }
}
```

The second method highlighted in Figure 10 decompiles as follows:

```
.method private hidebysig newslot virtual final instance
  int32 java.lang.Comparable/(Ljava.lang.Object;)IcompareTo(object) cil managed
{
    .override [mscorlib]System.IComparable::CompareTo

     return this.compareTo(obj1);
}
```

6 Transformations for non-interface types

The `(!classFile.IsInterface)` case starts at line 3818 in `DynamicTypeWrapper.cs`.

Two of the transformations for this case were already covered (in Sec. 5.1 and in Sec. 5.2). Two others remain:

- lines 3855 to 3858: conditionally call `AddUnsupportedAbstractMethods()` (Sec. 6.1)
- lines 3859 to 3885: for each method, check if adding an override is needed. Finally, `AddAutomagicSerialization(wrapper)` (Sec. 6.2)

which are covered in the following subsections.

6.1 `AddUnsupportedAbstractMethods()`

```
private void GenerateUnsupportedAbstractMethodStub(MethodBase mb)
{
  ParameterInfo[] parameters = mb.GetParameters();
  Type[] parameterTypes = new Type[parameters.Length];
  for (int i = 0; i < parameters.Length; i++)
  {
    parameterTypes[i] = parameters[i].ParameterType;
  }
  MethodAttributes attr =
    MethodAttributes.NewSlot | MethodAttributes.Virtual | MethodAttributes.Private;
  MethodBuilder m = typeBuilder.DefineMethod("__<unsupported>" + mb.DeclaringType.FullName + "/" + mb.Name,
                                             attr,
                                             ((MethodInfo)mb).ReturnType,
                                             parameterTypes);

  CodeEmitter.Create(m).EmitThrow("java.lang.AbstractMethodError",
              "Method " + mb.DeclaringType.FullName + "." + mb.Name + " is unsupported by IKVM.");

  typeBuilder.DefineMethodOverride(m, (MethodInfo)mb);
}
```

6.2 `AddAutomagicSerialization(wrapper)`

Lines 78 to 133 in `Serialization.cs`. It handles four cases:

- the received type wrapper is a Java enum. Add the `SerializableAttribute` to the output type.
- the wrapper subtypes `java.io.Externalizable`. In this case,

```
MethodWrapper ctor = wrapper.GetMethodWrapper("<init>", "()V", false);
if (ctor != null && ctor.IsPublic)
{
  MarkSerializable(wrapper);
  ctor.Link();
  serializationCtor = AddConstructor(wrapper.TypeAsBuilder, ctor, null, true);
  if (!wrapper.BaseTypeWrapper.IsSubTypeOf(serializable))
  {
    AddGetObjectData(wrapper);
  }
  if (wrapper.BaseTypeWrapper.GetMethodWrapper("readResolve",
                                               "()Ljava.lang.Object;",
                                               true) != null)
  {
    RemoveReadResolve(wrapper);
  }
}
```

- the wrapper subtypes `java.io.Seriallizable`. In this case,

```
ConstructorInfo baseCtor = wrapper.GetBaseSerializationConstructor();
if (baseCtor != null && (baseCtor.IsFamily || baseCtor.IsFamilyOrAssembly))
{
  MarkSerializable(wrapper);
  serializationCtor = AddConstructor(wrapper.TypeAsBuilder, null, baseCtor, false);
  AddReadResolve(wrapper);
}
```

- none of the above. TODO.

7 FYI

7.1 `java_lang_Object`

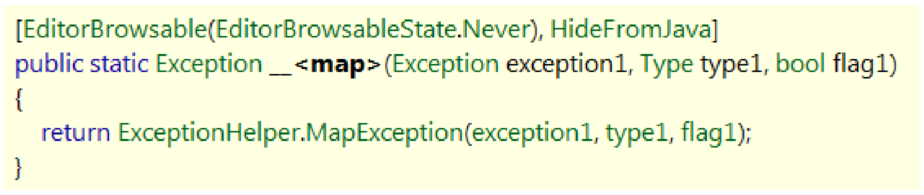
Quoting:

Remapped types extend their alter ego [e.g. the type wrapper `cli.System.Object` *is a remapped type] (e.g. cli.System.Object must appear to be derived from java.lang.Object) except when they're sealed, of course.*

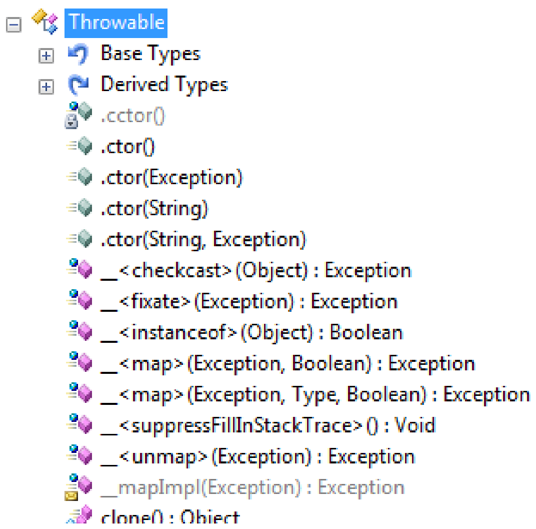
Note: invoking `GetWrapperFromType(Type type)` with `type` being `System.Object` returns `CompiledRemappedTypeWrapper[java.lang.Object]`. However, that result is changed when the aforementioned method is invoked from `GetWrapperFromAssemblyType(Type type)`: in that case a `DotNetTypeWrapper[cli.System.Object]` is finally returned (its `BaseTypeWrapper` however points to `CompiledRemappedTypeWrapper[java.lang.Object]`). Summing up:

- The wrapper for the base type of `System.Object` is the wrapper of `java.lang.Object`.
- Under that light, we get to see what is meant by: *cli.System.Object must appear to be derived from java.lang.Object.*

```
static Compiler()
{
  getTypeFromHandleMethod =
    typeof(Type).GetMethod("GetTypeFromHandle",
                           BindingFlags.Static | BindingFlags.Public,
                           null,
                           new Type[] { typeof(RuntimeTypeHandle) },
                           null);

  /* similarly for:
       monitorEnterMethod, nik for: System.Threading.Monitor.Enter
       monitorExitMethod , nik for: System.Threading.Monitor.Exit
       keepAliveMethod  , nik for: System.GC.KeepAlive
  */

  java_lang_Object     = CoreClasses.java.lang.Object.Wrapper;
  java_lang_Throwable  = CoreClasses.java.lang.Throwable.Wrapper;
  java_lang_Class       = CoreClasses.java.lang.Class.Wrapper;

  java_lang_ThreadDeath = ClassLoaderWrapper.LoadClassCritical("java.lang.ThreadDeath");

  cli_System_Object    = DotNetTypeWrapper.GetWrapperFromDotNetType(typeof(System.Object));
  cli_System_Exception = DotNetTypeWrapper.GetWrapperFromDotNetType(typeof(System.Exception));


  // HACK we need to special case core compilation, because the __<map> methods are HideFromJava
  if(java_lang_Throwable.TypeAsBaseType is TypeBuilder)
  {
    . . .
  }
  else
  {
    mapExceptionMethod =
      java_lang_Throwable.TypeAsBaseType.GetMethod("__<map>",
                                        new Type[] { typeof(Exception), typeof(Type), typeof(bool) });
    /*- method
          __<map>(System.Exception, System.Type, Boolean)
        is declared in
          java.lang.Throwable
        and returns a
          System.Exception
        (see screen capture below)
      */

    /*- similarly for
         mapExceptionFastMethod,
         suppressFillInStackTraceMethod,
         unmapExceptionMethod,
         fixateExceptionMethod */
  }

  getClassFromTypeHandle =
     ClassLoaderWrapper.LoadClassCritical("ikvm.runtime.Util")
    .GetMethodWrapper("getClassFromTypeHandle",
                      "(Lcli.System.RuntimeTypeHandle;)Ljava.lang.Class;",
                      false);
  getClassFromTypeHandle.Link();
}
```



```
[EditorBrowsable(EditorBrowsableState.Never), HideFromJava]
public static Exception __<map>(Exception exception1, Type type1, bool flag1)
{
   return ExceptionHelper.MapException(exception1, type1, flag1);
}
```

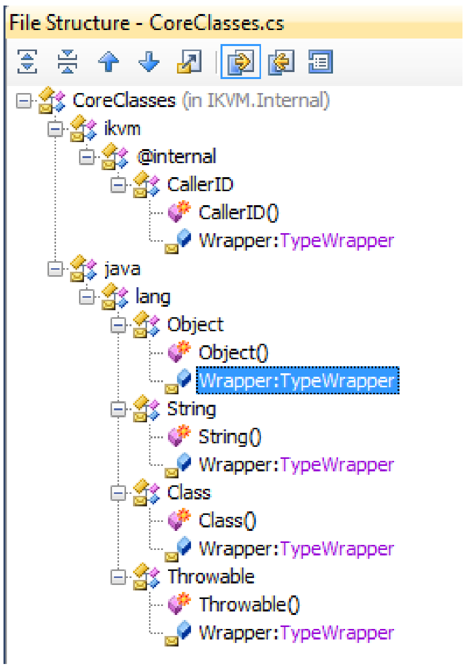


```
Throwable
  Base Types
  Derived Types
  .cctor()
  .ctor()
  .ctor(Exception)
  .ctor(String)
  .ctor(String, Exception)
  __<checkcast>(Object) : Exception
  __<fixate>(Exception) : Exception
  __<instanceof>(Object) : Boolean
  __<map>(Exception, Boolean) : Exception
  __<map>(Exception, Type, Boolean) : Exception
  __<suppressFillInStackTrace>() : Void
  __<unmap>(Exception) : Exception
  __mapImpl(Exception) : Exception
  clone() : Object
```

7.2 CoreClasses.java.lang.Object.Wrapper

`CoreClasses.java.lang.Object.Wrapper` (Figure 11) is loaded like this:

```
ClassLoaderWrapper.LoadClassCritical("java.lang.Object")
```


File Structure - CoreClasses.cs

```
CoreClasses (in IKVM.Internal)
  ikvm
    @internal
      CallerID
        CallerID()
        Wrapper:TypeWrapper
  java
    lang
      Object
        Object()
        Wrapper:TypeWrapper
      String
        String()
        Wrapper:TypeWrapper
      Class
        Class()
        Wrapper:TypeWrapper
      Throwable
        Throwable()
        Wrapper:TypeWrapper
```

Figure 11: `CoreClasses`, Sec. 7.2

7.3 Miranda methods

```
sun/tools/java/ClassDefinition.java:

    // In early VM's there was a bug -- the VM didn't walk the interfaces
    // of a class looking for a method, they only walked the superclass
    // chain. This meant that abstract methods defined only in interfaces
    // were not being found. To fix this bug, a counter-bug was introduced
    // in the compiler -- the so-called Miranda methods. If a class
    // does not provide a definition for an abstract method in one of
    // its interfaces then the compiler inserts one in the class artificially.
    // That way the VM didn't have to bother looking at the interfaces.
    //
    // This is a problem. Miranda methods are not part of the specification.
    // But they continue to be inserted so that old VM's can run new code.
    // Someday, when the old VM's are gone, perhaps classes can be compiled
    // without Miranda methods. Towards this end, the compiler has a
    // flag, -nomiranda, which can turn off the creation of these methods.
    // Eventually that behavior should become the default.
    //
    // Why are they called Miranda methods? Well the sentence "If the
    // class is not able to provide a method, then one will be provided
    // by the compiler" is very similar to the sentence "If you cannot
    // afford an attorney, one will be provided by the court," -- one
    // of the so-called "Miranda" rights in the United States.

    /**
     * Add a list of methods to this class as miranda methods. This
     * gets overridden with a meaningful implementation in SourceClass.
     * BinaryClass should not need to do anything -- it should already
     * have its miranda methods and, if it doesn't, then that doesn't
     * affect our compilation.
     */
    protected void addMirandaMethods(Environment env,
                                     Iterator mirandas) {
        // do nothing.
```

Quoting from `http://weblog.ikvm.net/CommentView.aspx?guid=65`:

```
abstract class Foo implements Runnable {
    Foo() { run(); }
}
```

> *jikes doesn't generate Miranda methods. Take the above class, for example. This compiles (and is legal). When javac compiles this code, it inserts an public abstract void run() method into Foo, this method is called a Miranda method. Older VMs (probably in the 1.0 time frame) didn't think the class was valid without this method. Jikes doesn't do this and IK.VM.NET couldn't handle that. Fixed.*

From `http://weblog.ikvm.net/PermaLink.aspx?guid=2bb118cf-4a21-4494-a50f-59177a1c8af1`

> *Fixed a Miranda bug that caused incorrect classes to be generated for abstract classes implementing java.lang.Comparable but not implementing compareTo.*

Listing 7: See Sec. 5.1.5

```
private void SetupGhosts(IKVM.Internal.MapXml.Root map)
{
  ghosts = new Dictionary<string, List<TypeWrapper>>();

  // find the ghost interfaces
  foreach(IKVM.Internal.MapXml.Class c in map.assembly.Classes)
  {
    if(c.Shadows != null && c.Interfaces != null)
    {
      // NOTE we don't support interfaces that inherit from other interfaces
      // (actually, if they are explicitly listed it would probably work)
      TypeWrapper typeWrapper = GetLoadedClass(c.Name);
      foreach(IKVM.Internal.MapXml.Interface iface in c.Interfaces)
      {
        TypeWrapper ifaceWrapper = GetLoadedClass(iface.Name);
        if(   ifaceWrapper == null
              /*- e.g., in IKVM there's no type at all for Serializable */
           || !ifaceWrapper.TypeAsTBD.IsAssignableFrom(typeWrapper.TypeAsTBD)
             /*- e.g. !java_lang_CharSequence.IsAssignableFrom(System.String) */
           )
        {
          AddGhost(iface.Name, typeWrapper);
        }
      }
    }
  }
  // we manually add the array ghost interfaces
  /*- every non-null Java array is serializable and cloneable */
  TypeWrapper array = ClassLoaderWrapper.GetWrapperFromType(typeof(Array));
  AddGhost("java.io.Serializable", array);
  AddGhost("java.lang.Cloneable", array);
}


private void AddGhost(string interfaceName, TypeWrapper implementer)
{
  List<TypeWrapper> list;
  if(!ghosts.TryGetValue(interfaceName, out list))
  {
    list = new List<TypeWrapper>();
    ghosts[interfaceName] = list;
  }
  list.Add(implementer);
}


internal TypeWrapper[] GetGhostImplementers(TypeWrapper wrapper)
{
  List<TypeWrapper> list;
  if (!ghosts.TryGetValue(wrapper.Name, out list))
  {
    return TypeWrapper.EmptyArray;
  }
  return list.ToArray();
}
```