Master Thesis:

Java Sourcecode to Eiffel Sourcecode Compiler

Marco Trudel

Submitted on December 29. 2008 at ETH Zürich

Prof. Dr. Bertrand Meyer Dr. Till Bay

Contents

1 Preface

Java and Eiffel are modern programming languages both having their advantages and disadvantages. One particular advantage-disadvantage pair led to the idea of doing this master thesis:
For Java, one of the big advantages is that a lot of well built libraries exist that have proven themselves and are in widespread use. One of the big disadvantages is that applications built to be distributed to end-users suffer the dependency of a Java Runtime Environment (JRE). An average computer user does not know what a JRE is, what version is installed (if one at all) or how to upgrade (install) it.
Eiffel on the other hand has exactly the opposite situation: It is a well built language with enormous potential. But to become popular, an excellent foundation of libraries covering at least a big part of the everyday needs has to exist. Currently, Eiffel has a gap in this regard. But it doesn't suffer a runtime dependency; compiled applications are binaries.

A Java sourcecode to Eiffel sourcecode compiler allows having the best of these both worlds:

- Java libraries can be compiled to Eiffel and being used in Eiffel projects. This provides Eiffel with the missing foundation of good libraries.
- Java applications can be compiled to binaries with no runtime dependency by compiling them to Eiffel and then compile the Eiffel project.

Of course the second point also opens the quite interesting possibility of switching the programming language to Eiffel in a project that is being done in Java. So that development can continue in Eiffel.
With Eiffel being the language with built-in support for quality (Design by contract and other unique new O-O principles like "command/query separation" or "uniform access"), this allows bringing Java projects that need to improve quality or maintainability to Eiffel.

2 Results

In the scope of this Master Thesis, a standalone compiler with a GUI has been implemented that compiles all Java language constructs to Eiffel and provides a runtime environment that is able to handle SWT applications. With this compiler, at least simple Java SWT applications can be compiled to Eiffel and will execute when the Eiffel code is compiled.

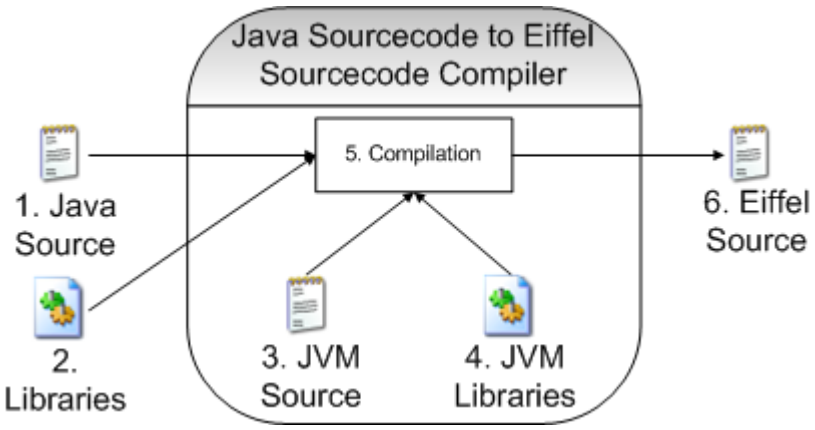
Three main problems have been discovered while designing and implementing the compiler:

- Java has less restrictive naming conventions for fields and methods than Eiffel. For instance Java is case sensitive and allows equal names for fields, methods and method arguments. Also, words that are often used as names in Java are reserved keywords in Eiffel ("result" or "from" for instance). These points raise the necessity to decorate field, method and method argument names for Eiffel.
- In Java, classes are highly interconnected. For instance a simple HelloWorld application uses java.lang.System (for System.out.println("HelloWorld")), System then uses java.lang.SecuritManager which uses java.awt.AWTPermission and this way, a simple HelloWorld already pulls in big parts of the AWT and Swing code. This leads to quite huge binaries (~36mb for a simple HelloWorld).
- A lot of Java libraries depend on the Java way of threading (for instance for synchronizing or waiting/notifying) and a properly set up runtime (for instance the presence of "jvm.dll", "java.dll" and others). Therefore, using Java libraries in Eiffel code requires the whole code to use the Java way of threading and a specific starting point for the execution (to set up the Java runtime) as well as some resources, as the runtime libraries, to be present.

These three points lead to the conclusion that using Java code in Eiffel is - although completely possible - not really feasible. Even if decorating names will be highly optimized and the resulting amount of pulled in Java classes reduced to a minimum (discussed in the Open Issues chapter), the three problems will remain.
For compiling Java code to binaries on the other hand, the only problem is the size of the resulting binaries. Considering that the only two products on the market providing Java to native compilation (Excelsior JET [1] and GCJ [2]) create binaries with comparable sizes, this can be tolerated. Also, there is potential for optimization as will be discussed in the Open Issues chapter.

3 The overall compiler design

This diagram shows how the compiler works:



1. The user provides the source code of the project he wants to compile (e.g. a class showing a SWT Window and the source code of the SWT library).
2. If required, the user also provides JNI libraries used by the provided source code (e.g. “swt-win32-3349.dll” for SWT).
3. The compiler then adds all referenced classes from the JVM library so that the project can be compiled completely.
4. The compiler also uses all JVM libraries to correctly map all native functions in the JVM classes.
5. With this information, the compiler creates the complete Eiffel source code with all native methods statically mapped to the libraries.
6. The created source code can now either be compiled with an Eiffel compiler (if the user input is an application and not a library) or used in an Eiffel application.

Step 3 and 4 are not done anew each time something will be compiled. Since the Eiffel code of the JVM source is static, it is bundled with the compiler and a precompile will be built the first time the user compiles something. By reusing this precompile, later compilations will be a lot faster.

4 Java Language Constructs

All Java source files provided from the user as described in chapter 3 are compiled to Eiffel source code one at a time. For each file, the abstract syntax tree (AST) is generated and the compiler is doing multiple transformations (steps) on it. The basic idea behind the stepping is that a big part of the Java language constructs have a direct counterpart in Eiffel (e.g. an if-statement). However, some language constructs do not have that and therefore are transformed into equal Java code with an Eiffel counterpart first. The overall Java sourcecode to Eiffel sourcecode works this way:

1. Generate the AST of a Java source file
2. Perform multiple transformations on the AST to create Java constructs that have an Eiffel equivalent. E.g. Create full classes from anonymous/inner classes or transform extended for loops to ordinary for loops.
3. Generate the Eiffel equivalent from the transformed AST and save the generated file(s).

This approach with multiple transformations in step two leads to small specific transformers and makes the code clean and maintainable. The following subchapters describe all transformers in the order they are applied. All Java language constructs are processed in at least one of them.

4.1 Creating the AST

The Eclipse Java development toolkit (JDT) is used to create the AST from Java source files. It has been chosen since it is a very mature product with a lot of utilities and sophisticated features. One of the most important features used in the compiler is the possibility of automatic binding information resolving. With it, AST nodes will carry additional information about their purpose. For instance a node that represents the invocation of a method carries information about that method and its defining class. A node representing the usage of a variable carries information about that variable (for instance where it is defined or what its type is).

4.2 OptimizationsTransformer

The compiler used to create the AST (see chapter 4.1) has a useful feature: It performs optimizations on statements that can be simplified or otherwise enhanced. One of the optimization that is being done is constant value expression calculation. For instance in

```
final int i = 5;
short s = i * 2;
```

the compiler adds the constant value of 10 as information to the AST node for “i * 2”.

This transformer replaces nodes with the corresponding constant (e.g. “i * 2” with “10” in this case). This allows circumventing the need of supporting downcasts as shown in the code above (i can only be assigned to s because the value is known to fit into a short). In Eiffel, this calculation would, without the transformation, result in a compiler error.

4.3 GenericRemover

Java generics can do much more than Eiffel generics. Only keeping the ones that are supported in Eiffel has shown to be quite complex and too time consuming.

Therefore, I have decided to completely transform all generic code to the non-generic counterparts.

The changes consist of adapting…

… generic types (classes and interfaces):

```
class TestClass<T, A, B extends String> {
   public void foo(T t) { }

}
```

becomes

```
class TestClass {
   public void foo(Object t) { }
}
```

… generic methods:

```
public <T> T foo(String s) { … }
```

becomes

```
public Object foo(String s) { … }
```

… parameterized types:

```
java.util.List<String> l = …;
```

becomes

```
java.util.List l = …;
```

Of course this also requires that all code working with generic types and methods need the correct casts. Changing such code consists of adapting…

… method invocation expressions:

```
public <T extends String> void foo(T t) {
   t.equalsIgnoreCase("a");
}
```

becomes

```
public void foo(Object t) {
   ((String)t).equalsIgnoreCase("a");
}
```

… method result types:

```
public <T> T foo() { … }

public void bar() {
   String s = foo();
}
```

becomes

```
public Object foo() { … }

public void bar() {
   String s = ((String)foo());
}
```

There are a couple of other cases where casts are needed. But since no OpenJDK and SWT class contains these cases, it has not (yet) been implemented. Doing so will be straightforward.

4.4 JavaDocRemover

Javadoc comments are removed since it seems that the created Eiffel code will not be reused by developers. It will be easy to transform Javadoc comments to Eiffel comments in case this feature will once be needed.

4.5 NoBlockTransformer

Multiple transformations depend on the presence of a block ({}) to determine the parent statement for the transformation. But in Java, multiple language constructs allow omitting blocks if there's only one statement. Therefore, e.g.:

```
if(…) fooBar();
```

is transformed to

```
if(…) { fooBar(); }
```

4.6 ContinueTransformer

Later transformations will transform switch statements and labeled blocks to loops (see below). Since these two language constructs support “break”, “break” remains valid and will point to the loop that has been created for the previous language construct. “continue” on the other hand is not supported by these two, leading to point to the wrong loop construct if it’s not having a label:

```
loop:
for(int i = 1; i < 5; i++) {
   switch(i) {
      case 1: continue;      // belongs to the loop
      case 2: continue loop; // belongs to the loop
      case 3: break;         // belongs to the switch
      case 4: break loop; // belongs to the loop
   }

   aBlock: {
      if(i == 1) continue;      // belongs to the loop
      if(i == 2) continue loop; // belongs to the loop
      if(i == 3) break;         // belongs to the labeled block
      if(i == 4) break loop; // belongs to the loop
   }
}
```

would, without proper adaption, become:

```
loop:
for(int i = 1; i < 5; i++) {
   for(…) { // transformed switch, see below
      if(i == 1) continue;      // belongs to the "switch"
      if(i == 2) continue loop; // belongs to the loop
      if(i == 3) break;      // belongs to the "switch"
      if(i == 4) break loop; // belongs to the loop
   }

   for(…) { // transformed labeled block, see below
      if(i == 1) continue;      // belongs to the "labeled block"
      if(i == 2) continue loop; // belongs to the loop
      if(i == 3) break;      // belongs to the "labeled block"
      if(i == 4) break loop; // belongs to the loop
   }
}
```

Therefore, this transformer adapts such “continue” statements to always use a label. If the enclosing loop does not yet have one, it is created.

4.7 AssignTransformer

In Java, assignments of basic types can be of the form

```
a += b;
```

This transformer adapts these non-standard assignments to

```
a = a + b;
```

There is one special case: Code that is automatically downcasted like

```
int i = 5;
long l = Long.MAX_VALUE;
i += l;
```

Of course, the standard equivalent

```
int i = 5;
long l = Long.MAX_VALUE;
i = i + l;
```

is illegal. Such cases need the correct cast and are therefore transformed to:

```
int i = 5;
long l = Long.MAX_VALUE;
i = (int)(i + l);
```

4.8 StaticImportTransformer

This transformer adapts usages of non-qualified static fields and methods that have been statically imported to use the proper qualifier.

Considering this class:

```
package tmp;

class StaticClass {
   public static int aField;
   public static void aMethod() { }
}
```

It allows this code:

```
import static tmp.StaticClass.*;

class TestClass {
   public void aMethod() {
      int i = StaticClass.aField; // ok
      tmp.StaticClass.aMethod(); // ok
      int j = aField;
      aMethod();
   }
}
```

Such usages of static imports become:

```
class TestClass {
   public void aMethod() {
      int i = StaticClass.aField; // ok
      tmp.StaticClass.aMethod(); // ok
      int j = StaticClass.aField;
      StaticClass.aMethod();
   }
}
```

4.9 EnumerationTransformer

Enumerations are nothing else than synthetic sugar for classes extending from java.lang.Enum. Therefore, enumerations like

```
enum Test
{
   Foo, Bar;
   public void aMethod() { }
}
```

are transformed to

```
abstract class Test extends java.lang.Enum {
   public static Test Foo;
   public static Test Bar;
   public void aMethod() { }

   private Test(String arg0, int arg1) {
      super(arg0, arg1);
   }
   public static Test[] values() {
      return null;
   }
   public static Test valueOf(String s) {
      return null;
   }
}
```

The methods values() and valueOf() as well as the initialization of the enum constants are not yet done since it's never used yet. Doing so will be quite easy.

4.10 AnnotationTransformer

Annotations are nothing else than synthetic sugar for interfaces extending from java.lang.annotation.Annotation. Therefore, annotations like

```
@interface MyAnnotationType {
    int someValue();
    String someOtherValue();
}
```

are transformed to

```
interface MyAnnotationType {
   int someValue();
   String someOtherValue();
}
```

Extending java.lang.annotation.Annotation has not yet been done since it's never used yet. Doing so will be quite easy.

4.11 AnonymousClassTransformer

Anonymous classes are transformed to nested classes:

```
class Tmp {
   Object o = new Object() { };
   public void bar() {
      new Object() { };
   }
}
```

becomes

```
class Tmp {
   class TMP_1 extends Object { }
   Object o = new TMP_1();
   public void bar() {
      class TMP_2 extends Object { }
      new TMP_2();
   }
}
```

The classes use the same names as created from a Java to Bytecode compiler. The reason they are kept at their position and are not already separated is that they might access fields from the enclosing class(es) and method(s). This will be transformed in later steps.

4.12 ConstructorChainingTransformer

Constructor chaining is done implicitly in Java. Therefore this transformer adds all the necessary code:

- It adds "`extends Object`" to Objects not defining a superclass (except java.lang.Object).
- It adds the default constructor for Objects that do not define a constructor.
- It adds "super()" calls in all constructors that do not call another constructor (except java.lang.Object).

4.13 OuterClassAccessTransformer

Nested non-static classes can access outer (enclosing) classes. Since these nested classes will become standalone classes in Eiffel, they have no longer access to enclosing instances. This transformer therefore adds a "j_outer" field to the class that is a reference to the enclosing class. All constructors are changed so that the first

argument is the enclosing class and that “j_outer” is set. All accesses to fields and methods of outer classes are then transformed to have the “j_outer” qualifier.

For example these classes:

```
class Outer {
   public int i;
   public void outerMethod() { }
   public void bar() { }

   class Inner {
      public void bar() {
        int j = i;
        outerMethod();
        Outer.this.bar();
      }
   }
}
```

become (including previous transformations):

```
class Outer extends Object {
   public int i;
   public Outer() {
      super();
   }
   public void outerMethod() { }
   public void bar() { }

   class Inner extends Object {
      private Outer j_outer;
      public Inner(Outer j_outer) {
         this.j_outer = j_outer;
         super();
      }
      public void bar() {
         int j = j_outer.i;
         j_outer.outerMethod();
         j_outer.bar();
      }
   }
}
```

Interesting here is that the created code is actually illegal: “super()” is not the first statement in the constructor of “Inner”. But this is required since the user might have provided a super constructor invocation containing access to a member of the enclosing class as argument (e.g. “super(i);” in this example). So the illegal Java code is by intention and later in the created Eiffel code, it’s perfectly legal.

Of course, creating instances of such nested classes now needs to be adapted:

```
new Inner();
```

becomes:

- if called in “Outer”:

```
new Inner(this);
```

- if called in “Inner”:

```
new Inner(j_outer);
```

Creating an instance from an independent class needs a qualifier:

```
new Outer().new Inner();
anOuter.new Inner();
```

these are changed to

```
new Inner(new Outer());
new Inner(anOuter);
```

This transformer is actually not as straightforward as presented here. There are a lot of special cases (e.g. multiple level of nesting, two nested classes at the same level instantiating each other and quite a few special cases more) which are not described. The interested reader is invited to consult the source (OuterClassAccessTransformer.java) to get detailed information.

4.14 OuterMethodAccessTransformer

Nested non-static classes can access final fields of enclosing methods. This transformer creates, for all accessed variables, a field in the nested class and adds the variable as constructor argument to initialize the field. This is a legal approach since the variables are final and therefore the passed references (or values for basic types) will always remain unchanged in the enclosing method.

For example, this class:

```
class Outer {
   public void bar(final int a, final int b) {
      new Object() {
         public void foo() {
            System.out.println(a > b);
         }
      };
   }
}
```

becomes (including previous transformations):

```
class Outer extends Object {
   public Outer() {
      super();
   }

   public void bar(final int a, final int b) {
      class Outer_1 extends Object {
         private int b;
         private int a;
         private Outer j_outer;

         public Outer_1(int b, int a, Outer j_outer) {
            this.b = b;
            this.a = a;
            this.j_outer = j_outer;
            super();
         }

         public void foo() {
            System.out.println(a > b);
         }
      }
      new Outer_1(b, a, this);
   }
}
```

Interesting here is that the created code is actually illegal: “super()” is not the first statement in the constructor of “Outer_1”. But this is required since the user might have provided a super constructor invocation containing variables of the enclosing method (e.g. “super(a, b);” in this example). So the illegal Java code is by intention and later in the created Eiffel code, it’s perfectly legal.

There is one special case: There might be multiple enclosing methods as in:

```
class Outer {
   public void bar(final int a) {
      new Object() {
         public void foo(final int b) {
            new Object() {
               public void foobar()
               {
                  if(a > b);
               }
            };
         }
      };
   }
}
```

Therefore, on processing, all variables from all enclosing methods until the outermost are considered.

4.15 ClassSeparator

After the previous transformers, nested classes do no longer depend on the enclosing classes and methods. Also, there are no longer Enumerations and Annotations.
This transformer therefore separates all classes. After that, only standalone non-nested interfaces and classes remain.

For instance these classes:

```
class Outer {
   enum AnEnum {
      Foo, Bar;
   };
   public void bar() {
      new Object() {
      };
   }
}
```

become (simplified):

```
class Outer extends Object {
   public void bar() {
      new Outer_1(this);
   }
}

abstract class Outer_AnEnum extends Enum {
   public static Outer_AnEnum Foo;
   public static Outer_AnEnum Bar;
}

class Outer_1 extends Object {
}
```

4.16 MiscellaneousTransformer

This transformer is a collection of minor changes that weren't worth being done in an own transformer. The changes are:

- removing empty statements:

```
if(true) { ; }
```

becomes

```
if(true) { }
```

This can be done now because the only situations where empty statements can't just be removed are language constructs with no block (since it is optional) and no content. For instance in

```
if(true);
```

the ";" is required. The "if" would otherwise consider the next statement as the body. But since the NoBlockTransformer above guarantees that all language constructs with an optional block do have a block, empty statements can safely be removed.

- moving extra dimensions in method declarations to the usual place:

```
public int bar()[][] { … }
```

becomes

```
public int[][] bar() { … }
```

- setting interface fields to "public static" if not done yet:

```
interface AnInterface {
   String s = "HelloWorld";
}
```

becomes

```
interface AnInterface {
   public static String s = "HelloWorld";
}
```

- changing string concatenation to use a StringBuffer:

```
"a" + 1 + "b"
```

becomes

```
new StringBuffer().append("a").append(1).append("b")
```

This way, java.lang.String can be handled just like an ordinary class without having to add a “+” infix operator to the created Eiffel class.

- removing unnecessary qualifiers in “this” expressions:

```
class TestClass {
   public void foo() {
      TestClass.this.foo();
   }
}
```

becomes

```
class TestClass {
   public void foo() {
      this.foo();
   }
}
```

4.17 AssertionTransformer

The compiler allows enabling or disabling assertions at compile time. If assertions are disabled, this transformer simply removes them from the code. Otherwise they are changed to ordinary if statements:

```
int a = …;
int b = …;
assert a == b;
assert a == b : "not equal";
```

becomes

```
int a = …;
int b = …;
if(!(a == b)) {
   throw new AssertionError();
}
if(!(a == b)) {
   throw new AssertionError("not equal");
}
```

4.18 AnnotationRemover

Annotations are actually nothing more than additional information for classes, fields and methods that can be queried at compile time and runtime. Since they are not needed for the own compilation and the runtime has only been implemented far enough to handle SWT applications, support for annotations has not yet been added. But, with the current manual mechanism for supporting reflection (described in the chapter about the runtime), adding support for annotations will be quite easy.

4.19 StandaloneBlockRemover

This transformer changes standalone blocks like

```
public void foo() {
    System.out.println("1");
    {
        int i = 1;
        System.out.println("2");
        System.out.println("3");
    }
    System.out.println("4");
    int i = 2;
}
```

to

```
public void foo() {
    System.out.println("1");
    int i = 1;
    System.out.println("2");
    System.out.println("3");
    System.out.println("4");
    int i = 2;

}
```

As shown in the example, this can lead to invalid code (multiple definitions of “i”). But name clashes like this have to be addressed by the final transformer that emits Eiffel code anyway, because Eiffel can only declare variables in the “local” block. Java on the other hand can declare them everywhere in different scopes of a method (for instance two for loops both having “i” as control variable).

4.20 LabeledBlockTransformer

Blocks can be aborted if they have a label. For instance:

```
public void foo(boolean doBreak) {
    System.out.println("1");
    myBlock: {
        System.out.println("2");
        if(doBreak) break myBlock;
        System.out.println("3");
    }
    System.out.println("4");
}
```

This transformer changes the code to (including all transformations):

```
public void foo(boolean doBreak) {
    System.out.println("1");
    myBlock:
    for(boolean tmp = true; tmp; tmp = false) {
        System.out.println("2");
        if(doBreak) { break myBlock; }
        System.out.println("3");
    }
    System.out.println("4");
}
```

So the problem is replaced by another problem that exists already (break/continue for loops) and will be handled later. This creates obviously an unnecessary overhead because of the loop. But in this case, I have favoured simplicity of the code generation over optimal generated code. Since the additional overhead is quite small

and this kind of code shouldn't be used at all in Java, the chosen solution should be fine.

4.21 FieldInitializationSeparator

In Java, fields can be directly initialized and it is exactly defined when the initializations - including initializers - will be executed. With the current ISE Eiffel version, fields can not be initialized directly. But also with the upcoming new "attribute" feature, initializing is lazy and it would be at least very hard to get the correct order of execution especially when also considering initializers. So this transformer changes:

```
class TestClass extends Object {
   public int i = 1;
   public static int is = 2;
   {
      System.out.println("initializer 1");
   }
   static {
      System.out.println("static initializer");
   }
   {
      System.out.println("initializer 2");
   }
   public static int js = 3;
   public int j = 4;

   public TestClass()
   {
      super();
   }
}
```

to:

```
class TestClass extends Object {
   public int i;
   public static int is;
   public static int js;
   public int j;

   public TestClass() {
      e_static_init();
      super();
      e_object_init();
   }
   private void e_object_init() {
      i = 1;
      System.out.println("initializer 1");
      System.out.println("initializer 2");
      j = 4;
   }
   public static void e_static_init() {
      is = 2;
      System.out.println("static initializer");
      js = 3;
   }
}
```

So, all initializations are put in the correct order into methods (one for object and one for static initializations). These methods are then called in the constructor; the static initializer before the (super)constructor call and the object initializer after it.
Of course the created code is actually illegal: “super()” is not the first statement in the constructor. But this is required since the static initializer is called before creating the object. So the illegal Java code is by intention and later in the created Eiffel code, it’s perfectly legal.

The last transformer that emits Eiffel code will make e_static_init to a global once function since it will only be executed the first time an Object is accessed. Also, it will call e_static_init of the parent class before doing the actual initializations.

With all the transformations so far, all implicit inheritance mechanism like field initialization, constructor chaining and the inheritance structure have been made explicit.

4.22 QualifiedClassNameTransformer

Sometimes in Java code, classes are used with the absolute name:

```
public void bar() {
    java.util.List javaList;
    org.eclipse.swt.widgets.List swtList;

    javaList = new java.util.LinkedList();
    swtList = new org.eclipse.swt.widgets.List(…);
}
```

Since in Eiffel, classnames need to be unique (the last transformer that emits Eiffel code will create them), these absolute names can be removed. This facilitates later transformations. Therefore, the code is changed to:

```
public void bar() {
    List javaList;
    List swtList;

    javaList = new LinkedList();
    swtList = new List(…);
}
```

Obviously the code is now illegal since “List” is ambiguous. But this is actually no problem since the AST contains all necessary information so that for the Eiffel code, unique names can be created.

4.23 SwitchTransformer

At first thought, one might be tempted to directly map Java “switch” statements to Eiffel “inspect” statements. Unfortunately, these two have completely different semantics:

- In Java, if an executed branch does not have a “break”, all following branches will be executed until a “break” occurs or all branches have been executed. In Eiffel on the other hand, there is no “break” and executing a branch will automatically finish the inspect statement.
- In Java, not entering any branch and not having a “default” is just doing nothing. In Eiffel, this is considered a bug and an exception is raised.

Therefore, this transformer changes:

```
public void bar(int i) {
   switch(i) {
      case 1:
         System.out.println("a");
         break;
      case 2:
          System.out.println("b");
          break;
       default:
           System.out.println("Not a and not b.");
   }
}
```

to:

```
public void bar(int i) {
   for(boolean loop = true, matched = false; loop; loop = false) {
      if(matched || i == 1) {
         matched = true;
         System.out.println("a");
         break;
      }
      if(matched || i == 2) {
          matched = true;
          System.out.println("b");
          break;
      }
      System.out.println("Not a and not b.");
   }
}
```

So the problem is changed to an already existing problem (break/continue in loops) that a later transformer will address anyway.
“loop” is here for being able to put a loop around the switch statement that will be executed exactly once. “matched” is used to save whether a branch has been executed. If yes, all following branches will also be executed until a “break” occurs.

4.24 WhileTransformer

This transformer changes “while” and “do-while” loops to “for” loops to unify later transformations.

A while loop

```
int i = 10;
while(i > 3) {
   i--;
}
```

becomes

```
int i = 10;
for(; i > 3; ) {
   i--;
}
```

A do-while loop

```
int i = 10;
do
{
 i--;
} while(i > 3);
```

becomes

```
int i = 10;
for(boolean first = true; first || (i > 3); first = false) {
  i--;
}
```

4.25 EnhancedForLoopTransformer

This transformer changes enhanced “for” loops to ordinary “for” loops to unify later transformations.

Enhanced for loops with an array

```
int[] ia = …;
for(int i : ia) { … }
```

become

```
int[] ia = …;
int[] tmp_arr = ia;
for(int tmp_index = 0; tmp_index < tmp_arr.length; tmp_index++) {
   int i = tmp_arr[tmp_index];
   …
}
```

“tmp_arr” is used since the for loop could also look like “for(int i : methodThatReturnsAnIntArray())”. This would lead to the method being executed multiple times without saving the result first in “tmp_arr” and working with it.

Enhanced for loops with something iterable

```
java.util.List<String> sl = …;
for(String s : sl) { … }
```

become

```
java.util.List<String> sl = …;
for(java.util.Iterator tmp_itr = sl.iterator(); tmp_itr.hasNext(); ) {
   String s = (String)tmp_itr.next();
   …
}
```

4.26 ForLoopTransformer

This transformer changes “for” loops (the only loop construct that is left after the previous transformations) to create a usable basis with no special cases for later transformations. For instance:

```
for(int i = 0, j[] = new int[10]; i < 10; i++) {
   j[i] = i;
}
```

becomes

```
for (; i < 10; ) {
   {
      int i = 0, j[] = new int[10];
   }

   j[i] = i;

   {
      i++;
   }
}
```

The transformer moves the initializer into an own block at the beginning of the loop and the updaters into an own block at the end of the loop. This is of course no longer valid Java code, but necessary for later transformations:

The initializers are changed to ordinary code since, like in the example, it might later be splitted into multiple statements. For instance

```
int i = 0, j[] = new int[10];
```

might also be a field declaration and becomes in a later transformer:

```
int i = 0;
int[] j = new int[10];
```

This will be done because Eiffel does not support mixing array and non-array declarations. The problem now is that such code can no longer be hold as initializer in the “for” loop (it would create an invalid AST and lead to an Exception). So, because such code will be transformed anyway in a later step, “for” initializers are here transformed to ordinary code in order to unify later processing. The last transformer that emits Eiffel code will use this block in the “from” part and therefore the code will be legal again.

The updaters are changed to ordinary code since a later transformation of continue and break has to treat them in a special way (see below). It won’t be enough to just emit Eiffel code for the updaters at the end of the loop body.

4.27 DeclarationSeparator

Although it's possible in Eiffel to declare multiple variables at once like

```
a, b, c: STRING
```

Java has additional possibilities like

```
int[] a, b[], c;
```

that can’t be expressed in Eiffel and makes separation of the declarations necessary.

Therefore, this transformer changes

```
int i = 0, j = 5;
int[] a, b[], c;
```

to

```
int i = 0;
int j = 5;
int[] a;
int[] b[];
int[] c;
```

This also facilitates later transformations because it ensures that all declarations only declare exactly one variable (with or without initialization).

4.28 ContinueBreakReturnTransformer

Eiffel does not have a language construct to skip code. So I can only think of two approaches to support continue, break and return:

- Raising an Exception and skip all the remaining statements in the current method by catching it in the rescue clause and ignore it (of course loops would have to be moved into own methods and being restarted on “continue”)
- Adding booleans like, for instance “return_set”, that are set to true after a return statement and all following code will be put into a “if(!return_set) { … code … }”.

The first approach is ugly because it would be a horrible wrong use of exceptions. Also, exceptions are quite slow. A loop that repeats a couple of hundred times and mostly using “contine” would suffer greatly in performance. Another problem would be that the rescue block needs to call retry in order to ignore the exception. This would lead to additional code in the method body because it needs to do nothing if “retry” was called for “return” or “break”.
The second approach is better in terms of reasonable use of language constructs and execution speed, but the code would become horribly nested the more continue/break/return statements are present in the code.

I use the second approach since it is obviously (for me) the better approach in terms of using language constructs as they are intended for and also keeping the possibility for code optimizations for multiple levels of nesting.

So this transformer changes

- return statements

```
public int dummy_max(int i, int j) {
    if(i > j) return i;
    if(j > i) return j;
    return -1;
}
```

to

```
public int dummy_max(int i, int j) {
   boolean etmp_return_set_3 = false;
   boolean etmp_return_set_2 = false;
   boolean etmp_return_set_1 = false;
   if(i > j) {
      Result = i;
      etmp_return_set_1 = true;
   }
   if(!etmp_return_set_1) {
      if(j > i) {
         Result = j;
         etmp_return_set_2 = true;
      }
      if(!etmp_return_set_2) {
         Result = -1;
         etmp_return_set_3 = true;
      }
   }
}
```

Unfortunately, even such a little method already creates quite messy code. Worthwhile mentioning is that the first Eiffel related code has been created here: the result is assigned to the non-existent variable “Result”.

- continue statements

```
for(int i = 0; i < 10; i++) {
   if(i % 2 == 0) continue;
   if(i == 1 || i == 9) continue;
   System.out.println(i + " should be prim!");
}
```

to (without the MiscellaneousTransformer)

```
for(; i < 10; ) {
   {
      int i = 0;
      boolean etmp_continue_set_1 = false;
      boolean etmp_continue_set_2 = false;
   }

   if(i % 2 == 0) {
      etmp_continue_set_1 = true;
   }
   if(!etmp_continue_set_1) {
      if(i == 1 || i == 9) {
         etmp_continue_set_2 = true;
      }
      if(!etmp_continue_set_2) {
         System.out.println(i + " should be prim!");
      }
   }

   {
      i++;
      etmp_continue_set_1 = false;
      etmp_continue_set_2 = false;
   }
}
```

- break statements

```
for(int i = 0; i < 10; i++) {
   if(i % 2 == 0) break;
   if(i == 1 || i == 9) break;
   System.out.println(i + " should be prim!");
}
```

to (without the MiscellaneousTransformer)

```
for(; !etmp_break_set_2 && (!etmp_break_set_1 && (i < 10)); ) {
   {
      int i = 0;
      boolean etmp_break_set_1 = false;
      boolean etmp_break_set_2 = false;
   }

   if(i % 2 == 0) {
      etmp_break_set_1 = true;
   }
   if(!etmp_break_set_1) {
      if(i == 1 || i == 9) {
         etmp_break_set_2 = true;
      }
      if(!etmp_break_set_2) {
         System.out.println(i + " should be prim!");
      }
   }

   if(!etmp_break_set_2) {
      if(!etmp_break_set_1) {
         i++;
      }
   }
}
```

There is one special case that has not been shown here: labelled nested loops to allow breaking or continuing one of the enclosing loops instead the innermost one. Handling the updaters then becomes a little more complex, but basically it's working exactly like shown here. The interested reader is invited to see how it is implemented in the sourcecode.

The transformation done here has one obvious possibility for optimization: Not every continue, break and return occurrence needs an own boolean variable. They can be combined to a single one for one loop (in the case of continue/break) and one method (in the case of return).

Having seen these minimal examples already becoming quite hard to read, it can be imagined how a "real" method having multiple uses of continue/break/return will look like. But I think this is the best approach to the problem with the facilities offered by Eiffel.

4.29 TernaryConditionTransformer

This transformer changes short-hand if-else statements to ordinary if-else statements.

Code like

```
return (obj == null) ? null : obj.toString();
```

becomes (without the ContinueBreakReturnTransformer)

```
String tmp;
if((obj == null)) {
   tmp = null;
} else {
   tmp = obj.toString();
}
return tmp;
```

This unifies code and facilitates further processing.

4.30 ArrayDeclarationTransformer

This transformer changes array declarations like

```
int a[];
int[] b[];
```

to

```
int[] a;
int[][] b;
```

in order to unify code and facilitate further processing.

4.31 StringObjectizer

This transformer changes plain string usages like

```
System.out.println("Foo");
```

to create a java.lang.String first:

```
System.out.println(new String("Foo"));
```

This is required since java.lang.String will be handled as any other class in the Eiffel code. The last transformer that emits Eiffel code will handle the correct java.lang.String instantiation from something like `"Foo"` what otherwise would be an Eiffel STRING_8 (what of course is not accepted by java.lang.String constructors).

4.32 VariableInitializationSeparator

This transformer splits all variable declarations with an initialization to a declaration without an initialization and an assignment. Then all variable declarations (now all without initialization) are moved to the top of the method so that they can be added to the “local” part of the Eiffel code later.

For instance

```
public void foo() {
   System.out.println(1);
   String a;
   System.out.println(2);
   String b = "b";
   System.out.println(b);
}
```

becomes

```
public void foo() {
    String a;
    String b;
    System.out.println(1);
    System.out.println(2);
    b = "b";
    System.out.println(b);
}
```

As with previous transformers, this can create illegal code. For instance

```
if(true) { String s = "a"; }
if(true) { int s = 1;      }
```

becomes

```
String s;
int s;
```

at the start of the method. Such and a lot of other similar cases will be handled in the last transformer that emits Eiffel code.

4.33 BoxingTransformer

Support for Java's boxing/unboxing mechanism could be added either by adding conversion features to the resulting Eiffel classes or by changing the Java code to explicitly doing the boxing.
Since basic types are mapped to the standard Eiffel expanded types (e.g. int becomes INTEGER_32) and these can't be changed, the conversion features would need to be in the wrapper classes (e.g. java.lang.Integer). But with them, code like "Object o = 7" wouldn't be supported because java.lang.Integer isn't involved. Therefore I have chosen to change automatic boxing/unboxing to be explicit.

This transformer therefore changes code like

```
Integer intObj = 7;
int i = intObj;
```

to

```
Integer intObj = new Integer(7);
int i = intObj.intValue();
```

4.34 VarargsTransformer

Varargs are nothing else than syntactic sugar for a method taking an array of the declared type. Therefore, code like

```
public void foo(int... ints) { }

public void bar() {
    foo();
    foo(1);
    foo(1, 2);
    foo(new int[] { 1, 2 }); // will not be changed
}
```

is transformed to

```
public void foo(int[] ints) { }

public void bar() {
    foo(new int[] { });
    foo(new int[] { 1 });
    foo(new int[] { 1, 2 });
    foo(new int[] { 1, 2 }); // has not been changed
}
```

4.35 BasicTypeOperationsTransformer

Java basic types will eventually be mapped to default Eiffel expanded types (e.g. int becomes INTEGER_32, float becomes REAL_32, …). Since in Java, arithmetic operations have a lot of implicit casts which the corresponding Eiffel classes do not support, this transformer makes them explicit:

- In Java, all characters in infix expressions other than string concatenation are implicitly used as integer. Therefore

```
int i = 5 * 'a' + 'b';
```

becomes

```
int i = 5 * (int)('a') + (int)('b');
```

- All infix operations except shifting implicitly convert the operands to have the same size. Therefore

```
long l = 5 / 4l;
```

becomes

```
long l = (long)(5) / 4l;
```

- Shifting in Java automatically results in at least an integer. Therefore

```
short s = 4;
int i = s >> 5;
```

becomes

```
short s = 4;
int i = (int)(s) >> 5;
```

This of course doesn't matter in a standard assignment as in the example but is necessary in expressions where the result of the shift operation is used in another operation.

- Bit operations result in at least an integer. Therefore

```
short s1 = 4, s2 = 5;
int i = s1 | s2;
```

becomes

```
short s1 = 4, s2 = 5;
int i = (int)(s1 | s2);
```

This of course doesn't matter in a standard assignment as in the example but is necessary in expressions where the result of the bit operation is used in another operation.

- Integer and Characters can be assigned to each other. Therefore

```
char c = 64;
int i = 'a';
```

becomes

```
char c = (char)(64);
int i = (int)('a');
```

4.36 EiffelFileWriter

With all the transformations so far, only Java language constructs remain that can directly be emitted as Eiffel code. Also, code has been unified so that no two of the remaining language constructs will be translated to the same Eiffel construct. So, emitting Eiffel code is quite straightforward in this transformer.

Please note that only these language constructs are described that change in the Eiffel code. For instance a parenthesized expression “(expr)” is not described since the parentheses are exactly equal in the resulting Eiffel code.

4.36.1 Basic types

The type mapping from Java to Eiffel is straightforward since the corresponding Eiffel types have the equal sizes:

Java	Eiffel	C (Cecil; JNI)
boolean	BOOLEAN	EIF_BOOLEAN
char	CHARACTER_32	EIF_CHARACTER
byte	INTEGER_8	EIF_INTEGER_8
short	INTEGER_16	EIF_INTEGER_16
int	INTEGER_32	EIF_INTEGER_32
long	INTEGER_64	EIF_INTEGER_64
float	REAL_32	EIF_REAL_32
double	REAL_64	EIF_REAL_64

Worthwhile mentioning is probably that – although it had some special semantics before the transformations – java.lang.String is no basic type. It is an ordinary class having some syntactic sugar which was transformed away in order to make it compilable.

4.36.2 Class names

Eiffel class names are created from the fully qualified Java class names in these three steps:

1. replace all “_” with “_1”
2. replace all “.” with “_”
3. change it to uppercase

So, java.lang.String for instance becomes JAVA_LANG_STRING.

With this transformation, the uniqueness of the Eiffel name is guaranteed since Java package and class names

- can’t start with a number (step 1 ensures that step 2 doesn’t lead to ambiguity)
- are case in-sensitive (step 3 doesn’t change lead to ambiguity)

4.36.3 Variable names

Multiple differences between Java and Eiffel lead to the problem that variable names from Java are not necessarily unique or even legal in Eiffel:

- Java is case sensitive. So two variables “foo” and “FOO” are different. Eiffel on the other hand is case insensitive and can’t differentiate between the two example variables.
- In Java, blocks have an own scope. For instance two successive loops in a method can define the same variable (e.g. “String tmp” and “int tmp”). In Eiffel, there’s only one scope for a complete routine. Therefore the two example variables above collide.
- Eiffel has a lot of reserved words that are quite frequently used in Java code (loop, end, invariant, result, …).
- In Eiffel, attribute, routine and routine parameter names need to be unique in a class (except parameters of different routines, they can have equal names).

To have a deterministic, simple and elegant way to circumvent all these potential pitfalls, this naming is being used:
- The name of local variables (routine parameters and routine locals) start with “local”, the name of attributes with “field”.
- Then the index of the variable when it is defined in the code is appended. E.g. in a routine with two parameters (a, b) and three locals (c, d, e), the names would be “local0” and “local1” for the parameters. The locals would be named “local2”, “local3” and “local4”.
- Although the variables are already unique with the described naming scheme, they are not really usable to debug code or work with code. Therefore, a “_” and the original name are also appended.

Java fields have another problem that needs special consideration. In a super-/sub-class relation, fields do not overwrite but hide each other. So it is possible to have a parent class “A” with the field “String data” and a subclass “B” with the field “int data”. “B” can even access “String data” from “A” via “super.data”.
This is of course all not possible in Eiffel and it creates multiple problems: There’s no way to access an overwritten attribute and there’s no way to change the type of an overwritten attribute. Therefore the previously described naming schema needs a little addition for fields:
- additionally, a “_” and the name of the defining class is appended to the variable name.

This circumvents the problem of hiding instead of overwriting fields since all attributes are now unique in the Eiffel class.

This Java class

```
package foo.bar;

public class TestClass {
   public String s;
   public int i;

   public void aMethod(String s) {
      this.s = s;
      int j = 5;
      i = 2 * j;
   }
}
```

is transformed into an the Eiffel class “FOO_BAR_TESTCLASS” with the fields “field0_s_FOO_BAR_TESTCLASS”, “field1_i_FOO_BAR_TESTCLASS” and the locals “local0_s”, “local1_j”.

4.36.4 Method names

Eiffel routines have, like attributes compared to Java fields, some limitations compared to Eiffel methods:

- There’s no in-class method overloading
- Private methods are not visible in sub-classes and need to be independent (no overloading, no dynamic binding).
- Static methods do not overwrite static methods of superclasses, they hide each other (exactly like Java fields).

To circumvent these problems, the Eiffel routine names are created from Java method names in this manner:

- Names for constructors are “make”. Names of non-constructor methods are “method_” and the original method name.
- To address the lack of support for in-class method overloading, the simple name of the types of all the parameters are appended to the name. So a method “foo” with a String and an int as parameters will have the name “method_foo_from_String_pint”. All primitive parameter types start with a “p” since otherwise “java.lang.Long” and the primitive “long” would be ambiguous in Eiffel. Of course there’s a possibility that the created name is not unique if only the simple name of parameters is taken (e.g. aMethod(java.util.List l) and aMethod(org.eclipse.Swt.Widgets.List l) would both result in method_aMethod_from_List(…)). But since using the fully qualified name would lead to unusable routine names and the whole OpenJDK source does not have this case, only the simple name is used for now.
- If the method is a constructor, static or private, a “_” and the Eiffel class name of the defining class is appended to get the correct non-overloading respectively hiding effect.

This Java class

```
package foo.bar;

public class TestClass {
   public TestClass () { }
   public TestClass (int i) { }
   public static void foo() { }
   public void bar() { }
   private void bar(int i) { }
   public void bar(String s) { }
}
```

is transformed into an Eiffel class with these routines:

- make_FOO_BAR_TESTCLASS
- make_from_pint_FOO_BAR_TESTCLASS
- method_foo_FOO_BAR_TESTCLASS
- method_bar
- method_bar_from_pint_FOO_BAR_TESTCLASS
- method_bar_from_String

4.36.5 Helper classes

The created Eiffel code depends on a couple of Eiffel utility classes that are presented in this chapter.

4.36.5.1 JAVA_ARRAY [G]

Since arrays are a language construct in Java but an ordinary class (ARRAY) in Eiffel, JAVA_ARRAY fills the gap between these two approaches by providing all facilities needed by a Java array.
The usage of this class is presented in the following chapters about array operations.

4.36.5.2 JAVA_INTERFACE_PARENT

This is the parent for all Java interfaces. It extends java.lang.Object (interfaces are objects) and undefines all implemented methods from it (interfaces do not bring implemented methods). This avoids having to undefine all java.lang.Object methods in all Interface classes. Without undefining, each implemented interface would bring implementations of “equals()”, “clone()”, “toString()” and others. But because these are already present in the class that implements the interface (it has to extend from java.lang.Object), this would lead to “multiple definition” compilation errors.

4.36.5.3 JAVA_PARENT

This is the parent for all classes that have been compiled from Java to Eiffel (therefore also for all Interfaces because JAVA_INTERFACE_PARENT extends java.lang.Object). It brings access to

- the jni_environment
- exception data and mechanisms (e.g. procedure “throw()”)
- support for calling super-methods
- a “dev_null (arg: ANY)” for using values of expressions that are not actually used in the Java code but need to be used in Eiffel
- support for shifting, division and modulo (they have different behaviour in Eiffel and in Java and need to be adapted)
- utility queries like “create_eiffel_string (java_string: JAVA_LANG_STRING) : STRING_8” or “create_java_string (eiffel_string: STRING_8) : JAVA_LANG_STRING”

These routines are used frequently and are therefore in the parent of all Java classes. The descriptions of the routines are in the respective chapters below.

4.36.5.4 JAVA_TYPEHELPER [G]

This class brings two queries: casting and instanceof checking. The “cast” query throws a ClassCastException if a cast is invalid.
The usage of this class is presented in the following chapters about casting and instanceof checking.

4.36.5.5 JAVA_VARIABLE [G]

Variables (fields and method parameters/locals) in Java have some abilities that the Eiffel counterparts do not have:

- Values can be assigned to fields (Eiffel only supports this if a setter method is assigned)
- Method parameters can be changed (new values can be assigned to them)
- Assigning something to a variable returns the value

To overcome these Eiffel differences without having to create a lot of additional code (e.g. a setter procedure for each field), all variables are put into the TUPLE-like expanded class “JAVA_VARIABLE [G]”.
JAVA_VARIABLE has the attribute “item: G assign set” to read and set the actual variable. So

```
public void foo() {
   int i, j;
   i = 5;
   j = 3 * i;
}
```

becomes

```
method_foo is
   local
      local0_i: JAVA_VARIABLE [ INTEGER_32 ]
      local1_j: JAVA_VARIABLE [ INTEGER_32 ]
   do
      local0_i.item := 5;
      local1_j.item := 3 * local0_i.item;
   end
```

Wrapping all variables into JAVA_VARIABLE solves all the problems mentioned above:

- All fields allow assignments without a specific setter procedure
- Method parameters are also wrapped into a JAVA_VARIABLE and therefore allow assigning new values to them
- Because JAVA_VARIABLE not only has the procedure “`set (new_item: G)`” but also ”`set_and_return (new_item: G): G`” and “`return_and_set (new_item: G): G`”, assignments that need a return value can easily be done in Eiffel. This even supports otherwise unthinkable postfix expressions (e.g. “i++”) via “`return_and_set`” without big efforts.

There are other advantages of this solution that will be described later:

- static variables can be done as once routines because the JAVA_VARIABLE will remain “static” and the actual variable can be accessed just like a field.
- try-catch blocks will also make use of the special JAVA_VARIABLE mechanism in order to support passing all locals (including expanded classes) into the try block (procedure) and still have the same reference as in the actual routine.

4.36.6 Member visibility

In Java, fields and methods can be public, protected, default and private.
Public can be achieved by “feature {ANY}”. Protected and default on the other hand requires listing all classes in the package and (in the case of protected) all subclasses. Private hides the members from subclasses. This is not possible in Eiffel and therefore it is implemented by adapting the member name (see “… names” chapters above). For private, the own class and all inner classes need to be listed. The own class because Java’s private allows classes to access members of other instance, Eiffel’s {NONE} doesn’t. The inner classes since they have access to the private members of outer classes.
Since all visibility modifiers except public require setting up potentially long lists and because this would prevent precompiling of classes, I have decided to set all members to public (feature {ANY}).

4.36.7 Static methods and fields

The only difference between static methods and non-static methods is that static methods are only able to work with static fields (while non-static methods can work with both, static and non-static fields). Therefore, methods need no special attention. Static fields on the other hand exist only once for all instances and threads. Therefore, they have global once semantics in Eiffel.
As already indicated in the chapter about JAVA_VARIABLE, this little Java class

```
class SmallClass {
   public static int i = 10;
   public int j = 10;
}
```

would become (strongly simplified; only fields are shown)

```
class
      SMALLCLASS


feature {ANY} -- public fields

      field0_i_SMALLCLASS: JAVA_VARIABLE [ INTEGER_32 ] is
            indexing
                  once_status: global
            once
            end

      field1_j_SMALLCLASS: JAVA_VARIABLE [ INTEGER_32 ]

end
```

A class now wanting to use “j” needs an ordinary instance of SMALLCLASS. Accessing “i” on the other hand has to work without an instance and is therefore mapped to “(create {SMALLCLASS}).i.item”. This of course implies the presence of default_create what leads to bad code since this makes class invariants impossible. But another even greater problem exists that makes a slight adaption of the proposed mechanism necessary:

```
interface AnInterface {
   public static int i = 10;
}
```

“i” needs to be accessible without an object of any form. But since AnInterface will be transformed into a deferred class, this is not possible (a deferred class can’t be instantiated).

This leads to moving all static methods and fields to a new “static” class (e.g. SMALLCLASS_STATIC for the example above). The base “non-static” class then extends from the new “static” class.
This brings a proper separation between the two actually completely different topics while still remaining complete consistency:

- SMALLCLASS can still access all static members
- SMALLCLASS doesn’t need to provide default_create. It can have creation procedures as wished and therefore also created arbitrary invariants.
- By using “(create {SMALLCLASS_STATIC}).i.item”, all static fields can be accessed even if they come from an interface or abstract class.

- default_create can be overwritten in such “static” classes to do the initializations (see the creation of “e_static_init” in FieldInitializationSeparator).

SmallClass therefore becomes (only slightly simplified this time):

```
class
   SMALLCLASS

inherit
   SMALLCLASS_STATIC
   JAVA_LANG_OBJECT

create
   make_SMALLCLASS

feature {ANY} -- public fields

   field1_j_SMALLCLASS: JAVA_VARIABLE [ INTEGER_32 ]

feature {NONE} -- constructors

   make_SMALLCLASS is
      do
         e_static_init;
         make_JAVA_LANG_OBJECT;
         method_e_object_init_SMALLCLASS;
      end

feature {ANY} -- private methods

   method_e_object_init_SMALLCLASS is
      do
         field1_j_SMALLCLASS.item := 10;
      end

end
```

and

```
class
   SMALLCLASS_STATIC

inherit
   JAVA_LANG_OBJECT_STATIC

feature {ANY} -- public fields

   field0_i_SMALLCLASS: JAVA_VARIABLE [ INTEGER_32 ] is
      indexing
         once_status: global
      once
      end

feature {ANY} -- public methods

   e_static_init is
      indexing
         once_status: global
      once
         Precursor;
         field0_i_SMALLCLASS.item := 10;
      end

end
```

4.36.8 Classes and interfaces

Classes (abstract and non-abstract) and interfaces can directly be written in Eiffel (as already shown in the previous chapter). The full mapping is:

```
deferred class -- “deferred“ for interfaces and abstract classes
   CLASSNAME -- Eiffel classname as described above

inherit
   JAVA_LANG_OBJECT -- the superclass and all implemented interfaces
      undefine
         -- abstract classes can undefine methods of the superclass!
      redefine
         -- all redefined routines
      end

create
   -- constructors

feature {NONE}

   -- constructors

feature

   -- attributes and routines

end
```

4.36.9 MethodDeclaration

Three different kinds of methods have to be handled separately: native methods, abstract methods and the rest (normal methods):

4.36.9.1 abstract methods

The mapping is:

```
methodname (parameters): result_type is
      deferred
   end
```

4.36.9.2 normal methods

The mapping is:

```
methodname (parameters): result_type is
   local
      -- all parameters again as JAVA_VARIABLE (described below)
      -- all locals
   do
      -- method body
   end
```

Noteworthy here is that, as described earlier, all method parameters are wrapped into a JAVA_VARIABLE. But since Eiffel would require an explicit conversion in many cases for parameters defined as JAVA_VARIABLE, the original definition is kept with a temporary name, a local with the real name is created and initialized by assigning the parameter. For instance passing a String to a method expecting a JAVA_VARIABLE [JAVA_LANG_OBJECT] would fail without an explicit conversion. Therefore (because code has to be created anyway), the conversions are left to the compiler. For instance the method

```
public void foo(int i) { }
```

is emitted as this Eiffel code

```
method_foo_from_pint (tmp_local0_i: INTEGER_32) is
   local
      local0_i: JAVA_VARIABLE [ INTEGER_32 ]
   do
      local0_i.item := tmp_local0_i;
   end
```

4.36.9.3 native methods

In Java, native methods are mapped to the corresponding DLLs at compile time. It can be changed to be done at runtime too if eventually necessary, but for now, it is easier this way.
The implementation of the JNI environment is described in the chapter about the Java Runtime System. Here it's only important to know that JAVA_PARENT has a feature "jni_env: POINTER" that provides the necessary jni environment that is needed by native methods.
Native methods also need the current object or, if the method is static, the class of the method. Reflection is also described in the next chapter about the Java Runtime System. Here it's only important to know that every class has a procedure "get_class" that returns the JAVA_LANG_CLASS of the current object.

This said, a native method

```
public static native int identityHashCode(Object x);
```

becomes

```
method_identityHashCode_from_object_JAVA_LANG_SYSTEM
      (tmp_local0_x: JAVA_LANG_OBJECT) : INTEGER_32 is
   local
      java_class: JAVA_LANG_CLASS
   do
      java_class := get_class
      Result := Java_java_lang_System_identityHashCode (jni_env,
            $java_class, $tmp_local0_x)
   end

Java_java_lang_System_identityHashCode (env, this: POINTER;
      local0_x: POINTER) : INTEGER_32 is
   external
      "dllwin %"java.dll%" signature (EIF_POINTER, EIF_REFERENCE,
         EIF_REFERENCE) : EIF_INTEGER_32"
   alias
      "_Java_java_lang_System_identityHashCode@12"
   end
```

The first argument of a native method is the JNI environment and the second the current object (the class if the method is static). Therefore the jni_env, the (in this example) current class and all the parameters are passed to a new routine that is doing a DLL call. Basic parameters are passed by value, from references (objects) the pointer is passed.
The name of the routine is, in order to be unique, the JNI name of the method. The corresponding DLL can be determined at compile time by looking through all provided DLLs for the JNI name. The signature can be created from the original method and the alias is necessary because on Windows, DLL function names are decorated.

4.36.10 MethodInvocation

Method invocations are equal in Java and Eiffel.

```
foo.bar(1, 2);
```

becomes (if foo is the first local variable and bar is a public non-static method)

```
local0_foo.method_bar_from_pint_pint (1, 2);
```

4.36.11 FieldDeclaration

Please note that this chapter has mostly (but not yet complete) been implied by the chapter “Static methods and fields” above.

Non-static fields from classes like

```
class TestClass {
   public String myInfoString;
}
```

become this Eiffel code:

```
field0_myInfoString_TESTCLASS: JAVA_VARIABLE [ JAVA_LANG_STRING ]
```

Static fields from classes like

```
class TestClass {
   public static String myInfoString;
}
```

become this Eiffel code:

```
field0_myInfoString_TESTCLASS: JAVA_VARIABLE [ JAVA_LANG_STRING ] is
   indexing
      once_status: global
   once
   end
```

Fields from interfaces (interfaces only have public static fields) like

```
interface TestInterface {
   public static String myInfoString = …;
}
```

become this Eiffel code:

```
field0_myInfoString_TESTINTERFACE: JAVA_VARIABLE [ JAVA_LANG_STRING ] is
   indexing
      once_status: global
   once
      e_static_init_TESTINTERFACE
   end
```

The call to e_static_init_TESTINTERFACE is needed since field initializations of interfaces are done the first time a field of the interface is accessed. Static and non-static fields of classes on the other hand are initialized while instantiating a concrete object and therefore the transformations before the EiffelFileWriter have added all necessary calls in the correct places.

4.36.12 Local variable declarations

Method variables like

```
String myString;
```

are put into the local part of the routine and become

```
local1_myString: JAVA_VARIABLE [ JAVA_LANG_STRING ]
```

4.36.13 ArrayType

Array types (fields or locals) like

```
String[] myStrings;
```

become

```
local0_myStrings: JAVA_VARIABLE [ JAVA_ARRAY [ JAVA_LANG_STRING ] ]
```

4.36.14 Assignment

If the result of an assignment is not needed, the generated Eiffel code only has a different assignment symbol:

```
i = 2 * 5;
```

becomes (without the OptimizationsTransformer which would replace “2 * 5” by “10”):

```
local0_i.item := 2 * 5;
```

But if the result is needed, the circumstance that all variables are wrapped into a JAVA_VARIABLE helps to circumvent that Eiffel doesn’t support returning values on assignments:

```
j = (i = 2 * 5) + 10;
```

becomes (without the OptimizationsTransformer which would replace “2 * 5” by “10”):

```
local1_j.item := (local0_i.set_and_return (2 * 5)) + 10;
```

4.36.15 ClassInstanceCreation

Creating instances of classes like

```
new java.lang.Integer(15);
```

become

```
create { JAVA_LANG_INTEGER }.make_from_pint_JAVA_LANG_INTEGER (15)
```

The anonymous creation procedure is used for all class instance creations since Java allows constructs that are not possible in Eiffel with the ordinary usage of “create”. For instance “anotherObject.aField = new String(…)” can’t be done in Eiffel without the anonymous creation because “create anotherObject.aField.make (…)” is invalid.

4.36.16 IfStatement

The only thing to consider when writing the Eiffel equivalent to Java “if” statements is that Java actually does not have an else-if construct. For instance the NoBlockTransformer reveals that

```
public String compare(int a, int b) {
   if(a < b) {
      return "a < b";
   } else if(b < a) {
      return "b < a";
   } else {
      return "b == a";
   }
}
```

is nothing else than

```
public String compare(int a, int b) {
   if(a < b) {
      return "a < b";
   } else {
      if(b < a) {
         return "b < a";
      } else {
         return "b == a";
      }
   }
}
```

The possibility of “if” and “else” statements without a block ({ }) simulates the if-else construct in Java. But since Eiffel does have an else-if construct, the original form is restored for the Eiffel code and the above example becomes (simplified):

```
method_compare_from_pint_pint (local0_a: INTEGER_32;
      local1_b: INTEGER_32) : JAVA_LANG_STRING is
   do
      if local0_a.item < local1_b.item then
         Result := create_java_string ("a < b");
      elseif local1_b.item < local0_a.item then
         Result := create_java_string ("b < a");
      else
         Result := create_java_string ("b == a");
      end
   end
```

4.36.17 ForStatement

Loops

```
for(int i = 0; i < 10; i++) System.out.println(i);
```

become

```
from
   local0_i.item := 0;
until
   not (local0_i.item < 10)
loop
   (create { JAVA_LANG_SYSTEM_STATIC }).field3_out_JAVA_LANG_SYSTEM.
         item.method_println_from_pint (local0_i.item);

   -- updaters:
   local0_i.item := local0_i.item + 1;
end
```

The only thing that needs to be adapted is the negation of the loop condition. The Java loop is executed as long as it holds and the Eiffel loop until it no longer holds.

4.36.18 ExpressionStatements

ExpressionStatements are expressions that build complete statements. Since expressions always create a result, this means that the code then contains an unused value (e.g. “foo();” with the non-void method foo). Because of the command-query separation, the Eiffel compiler rejects unused values of expressions (why would one ask a question without listening to the answer?). Therefore, expression statements need to use the result. The used solution is that all expression statements are written like “dev_null(expr)”.

4.36.19 Literals

4.36.19.1 BooleanLiteral

Java’s “true” becomes “True” and “false” becomes “False”.

4.36.19.2 NumberLiteral

- A double becomes “({REAL_64} *value*)”
- A float becomes “({REAL_32} *value*)”
- A long becomes “({INTEGER_64} *value*)”
- An int is written just as it is in Java

These are all number literals since there is no way to write a literal for something smaller than an int. A byte for instance needs to be written as “(byte)5”.

4.36.19.3 NullLiteral

The “null” in Java becomes Eiffel's “Void”.

4.36.19.4 CharacterLiteral

Since characters are mapped to CHARACTER_32, a Java character becomes “({CHARACTER_32} value)”. For instance “‘a’” becomes “({CHARACTER_32} ‘a’)”. Eiffel uses a different encoding for certain characters. Here is the complete encoding transformation:

Java	Eiffel
\t	%T
\b	%b
\n	%N
\r	%R
\f	%F
\”	%”
\’	%’
\\	\
\uABCD (Unicode, e.g. ’\u0041’ for ’A’)*	%/0xABCD/ (e.g. ‘%/0x0041/’)
\ABC (Octal, e.g. ‘\101’ for ‘A’)*	%/0cABC/ (e.g. ‘%/0c101/’)
%	%%

* A, B, C, D ∈ { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }

4.36.19.5 StringLiteral

If the Java string literal does not contain Unicode and octal values, all its special characters are transformed according to the table above and it is written as create_java_string(“str”).
If the Java string literal contains Unicode or octal values, this can’t be done since a string literal will in Eiffel be in STRING_8. But 255 is the biggest value that is accepted by STRING_8. Therefore, if there are Unicode or octal values, the JAVA_LANG_STRING is created by changing the string literal to using the char[] constructor of java.lang.String. For instance “A\400C” becomes “new String(new char[] { ‘A’, ‘\400’, ‘C’ })” and is transformed to the according Eiffel code.

4.36.19.6 TypeLiteral

Type literals are class requests like “String.class”, “int.class” or “java.util.list.List.class”. Primitive ones are printed as an access to the corresponding TYPE field of the wrapper class (e.g. “int.class” is handled as “Integer.TYPE”). Non-primitive types are printed like an access to the static method “get_class” of that class. All static classes get a global once method “get_class” that creates a JAVA_LANG_CLASS with all the required information. This is further explained in the “Java Runtime System” chapter under “Reflection”. So, “String.class” for instance is printed as “(create { JAVA_LANG_STRING_STATIC }).get_class”.

4.36.20 ThisExpression

Java’s “this” becomes Eiffel’s “Current”.

4.36.21 PrefixExpression

Prefix expressions are an expression (e.g. a variable “i”) with a prexifed operation (e.g. “++i”, “~i”, …). Here the complete overview of all prefixes and their mapping to Eiffel:

- Plus (+), Minus (-)
They stay the same for Eiffel.

- Increment (++), Decrement (--)
They only apply to variables. Since all variables are wrapped into a JAVA_VARIABLE, the prefix expressions can easily be implemented with JAVA_VARIABLE.set_and_return (e.g. “++i” becomes “i.set_and_return (i + 1)”.

- Not (!)
It only applies to booleans and becomes “not”.

- Complement (~)
It only applies to numbers and becomes “bit_not” (e.g. “~i” becomes “i.bit_not”).

4.36.22 Postfix Expression

Postfix expressions (increment (e.g. i++) and decrement (e.g. i--)) only apply to variables. Since all variables are wrapped into a JAVA_VARIABLE, the postfix expressions can easily be implemented with JAVA_VARIABLE.return_and_set (e.g. “i++” becomes “i.return_and_set (i + 1)”.

4.36.23 InfixExpression

All infix expressions and their mappings are:

- Plus (+): Equal in Eiffel.
- Minus (-): Equal in Eiffel.
- Times (*): Equal in Eiffel.
- Less (<): Equal in Eiffel.
- Greater (>): Equal in Eiffel.
- Less or equals (<=): Equal in Eiffel.
- Greater or equals (>=): Equal in Eiffel.
- Equals (==): Becomes “=”.
- Not equals (!=): Becomes “/=”.
- Divide (/): Division is done in JAVA_PARENT.div because division by zero has to be handled specially: division by zero not from float or double result in a NullPointerException. Also, the C compiler rejects statements like “a / 0”. So for instance “a / b” becomes “div(a, b)”.
- Remainder (%): Modulo is done in JAVA_PARENT.mod because of modulo by zero handling and because EIF_REAL32 and 64 do not support modulo. It has to be programmed explicitly.
- Left shift (<<): Becomes “|<<”.
- Right shift signed (>>): Becomes “|>>”
- Right shift unsigned (>>>): This is done in JAVA_PARENT.right_shift_unsigned since Eiffel does not support it directly.
- And (&): Becomes “bit_and”. For instance “a & b” becomes “a.bit_and(b)”.
- Or (|): Becomes “bit_or”.
- Xor (^): Becomes “bit_xor”.

- Conditional or (||): Becomes “or else” (it only applies to booleans).
- Conditional and (&&): Becomes “and then” (it only applies to booleans).

4.36.24 ArrayCreation

An array creation becomes the creation of a JAVA_ARRAY:

```
new int[5];
new String[3];
```

becomes

```
create { JAVA_ARRAY [ INTEGER_32 ] }.make (5);
create { JAVA_ARRAY [ JAVA_LANG_STRING ] }.make (3);
```

Arrays with multiple dimensions

```
new int[2][3][4];
```

become

```
create { JAVA_ARRAY [ JAVA_ARRAY [ JAVA_ARRAY [ INTEGER_32 ] ] ] }.
      make_multi (<< 2, 3, 4 >>);
```

4.36.25 ArrayInitializer

Directly initializing an array like in

```
int[] ia = { 1, 2, 3 };
double[] da = new double[] { 1, 2, 3 };
```

becomes

```
local0_ia.item := (create { JAVA_ARRAY [ INTEGER_32 ] }.
      make_from_array (<< 1, 2, 3 >>));
local1_da.item := (create { JAVA_ARRAY [ REAL_64 ] }.
      make_from_array (<< 1, 2, 3 >>));
```

4.36.26 ArrayAccess

Since JAVA_ARRAY overloads the bracket operations “[]”, array accesses are almost equal in Java and Eiffel. The only difference is that in Eiffel, access to a multidimensional array has to be in brackets:

```
int[] a1 = …;
int i1 = a1[0];

int[][] a2 = …;
int i2 = a2[0][1];
```

becomes

```
local0_a1.item := …;
local1_i1.item := local0_a1.item[0];

local2_a2.item := …;
local3_i2.item := (local2_a2.item[0])[1];
```

4.36.27 InstanceofExpression

Instanceof tests

```
if(expr instanceof String) { … }
```

become

```
if
   (create { JAVA_TYPEHELPER [ JAVA_LANG_STRING ] }).is_instanceof (expr)
then
   …
end
```

4.36.28 CastExpression

Cast expressions for objects

```
Object o = …;
String s = (String)o;
```

become

```
local0_o.item := …;
local1_s.item := (create { JAVA_TYPEHELPER [ JAVA_LANG_STRING ] }).
                    cast (local1_o.item);
```

Casting primitive types like

```
int i = (int)1.2d;
```

are implemented by executing the corresponding queries. The above cast for instance becomes (without the OptimizationTransformer which would replace "(int)1.2d" with "1"):

```
local0_i.item := ({REAL_64} 1.2).truncated_to_integer_64.to_integer_32;
```

The exhaustive list of primitive casting operations is:

char to byte	->	variable.code.to_integer_8
char to short	->	variable.code.to_integer_16
char to int	->	variable.code
char to long	->	variable.code.to_integer_64
char to float	->	variable.code.to_real
char to double	->	variable.code.to_double
byte to char	->	variable.to_character_32
byte to short	->	variable.to_integer_16
byte to int	->	variable.to_integer_32
byte to long	->	variable.to_integer_64
byte to float	->	variable.to_real
byte to double	->	variable.to_double
short to char	->	variable.to_character_32
short to byte	->	variable.to_integer_8
short to int	->	variable.to_integer_32
short to long	->	variable.to_integer_64
short to float	->	variable.to_real
short to double	->	variable.to_double
int to char	->	variable.to_character_32
int to byte	->	variable.to_integer_8
int to short	->	variable.to_integer_16
int to long	->	variable.to_integer_64
int to float	->	variable.to_real
int to double	->	variable.to_double
long to char	->	variable.to_character_32
long to byte	->	variable.to_integer_8
long to short	->	variable.to_integer_16
long to int	->	variable.to_integer_32
long to float	->	variable.to_real
long to double	->	variable.to_double

float to char	->	variable.truncated_to_integer_64.to_character_32
float to byte	->	variable.truncated_to_integer_64.to_integer_8
float to short	->	variable.truncated_to_integer_64.to_integer_16
float to int	->	variable.truncated_to_integer_64.to_integer_32
float to long	->	variable.truncated_to_integer_64
float to double	->	variable.to_double
double to char	->	variable.truncated_to_integer_64.to_character_32
double to byte	->	variable.truncated_to_integer_64.to_integer_8
double to short	->	variable.truncated_to_integer_64.to_integer_16
double to int	->	variable.truncated_to_integer_64.to_integer_32
double to long	->	variable.truncated_to_integer_64
double to float	->	variable.truncated_to_real

4.36.29 ConstructorInvocation and SuperConstructorInvocation

Because all constructors have a unique name, calling a (super)constructor is handled like an ordinary method invocation:

```
public TestClass() {
   this("hello");
}

public TestClass(String msg) {
   …
}
```

becomes

```
make_TESTCLASS is
   do
      make_from_string_TESTCLASS (create_java_string ("hello"));
   end

make_from_string_TESTCLASS (tmp_local0_msg: JAVA_LANG_STRING) is
   do
      …
   end
```

4.36.30 SuperMethodInvocation

Supporting Java's way of calling an overwritten method is rather complex because Eiffel only supports calling the predecessor of the current routine via "Precursor". Considering this situation:

```
class TestSuper {
   public java.util.List foo() { … }
}

class TestClass extends TestSuper {
   public void bar() {
      super.foo();
   }
   public java.util.LinkedList foo() { … }
}
```

In Eiffel, there's no way of calling TestSuper.foo from TestClass.bar (unless dynamic binding is broken, what really is not an option). Therefore, the implemented solution is to call TestClass.foo() and letting it call the Precursor. The resulting code looks like this:

```
class
   TESTCLASS

inherit
   TESTCLASS_STATIC
   TESTSUPER
      redefine
         method_foo
      end

feature {ANY} -- public methods

   method_bar is
      do
         dev_null (({JAVA_UTIL_LIST} #?
               call_super_function (agent method_foo, $method_foo)));
      end

   method_foo : JAVA_UTIL_LINKEDLIST is
      do
         if supercall_data.call_super then
            if supercall_data.routine = $method_foo then
               supercall_data.call_super := False
               supercall_data.super_result := Precursor
            else
               dev_null (Precursor)
            end
         else
            … -- original method code here
         end
      end

end
```

The method TestClass.bar cannot call TestClass.foo directly since the result type of TestSuper.foo is not compatible (it might be an ArrayList). Therefore, calling a supermethod always happens by either calling “call_super_function” or “call_super_procedure” (from JAVA_PARENT). They then

- set the once per thread variable “supercall_data.call_super” to true
- invoke the intended method that has been passed as agent (this method resets “call_super” and sets the super_result).
- return the result as ANY (invoking “call_super_function” therefore needs to cast the result as seen in the example).

This means that all methods overwriting another method need to check whether the supermethod has to be invoked. This is quite ugly but the only way I can think of to support supermethod invocations.

There’s one more problem to consider: a method might be overwritten multiple times in an inheritance hierarchy. If a method is overwritten three times and the middle class calls “call_super_function”, dynamic binding actually lets the agent execute the third method. This method will now call the second method which in turn needs to call the first method. That’s why also the address of the method is passed to “call_super_function/procedure”; so that called routines know whether the Precursor is already the intended routine or not yet. Of course only the result of the intended routine can be saved (all subroutines will return Void).

4.36.31 SuperFieldAccess

Since all fields have a unique name and therefore no overwriting happens, accessing a superfield is not different from accessing a field of the current class.

4.36.32 TryStatement and CatchStatement

The only way to implement Java exceptions in Eiffel is by its exception mechanism. Since the only way to catch exceptions in Eiffel is in the rescue clause, all try blocks have to be an own procedure. In order to not get a lot of additional procedures and keep the code structure the way it was originally, try blocks are implemented as inline agents:

```
public void foo() {
   System.out.println("1");
   try {
      someMethod();
   } catch(IllegalArgumentException ex) {
      ex.printStackTrace();
   } catch(Exception ex) {
      ex.printStackTrace();
   } finally {
      System.out.println("2");
   }
   System.out.println("3");
}
```

becomes

```
method_foo is
   local
      local0_ex: JAVA_VARIABLE [ JAVA_LANG_ILLEGALARGUMENTEXCEPTION ]
      local1_ex: JAVA_VARIABLE [ JAVA_LANG_EXCEPTION ]
   do
      (create { JAVA_LANG_SYSTEM_STATIC }).field3_out_JAVA_LANG_SYSTEM.
            item.method_println_from_string (create_java_string ("1"));
      (agent (try1_0_ex:
            JAVA_VARIABLE [ JAVA_LANG_ILLEGALARGUMENTEXCEPTION ];
            try1_1_ex: JAVA_VARIABLE [ JAVA_LANG_EXCEPTION ])
         local
            try1_catch: BOOLEAN
         do
            if not try1_catch then
               -- try
               method_someMethod;
            else
               -- catch
               if ({JAVA_LANG_ILLEGALARGUMENTEXCEPTION} #?
                     current_thread_exception) /= Void then
                  try1_0_ex.item := {JAVA_LANG_ILLEGALARGUMENTEXCEPTION}
                        #? current_thread_exception;
                  try1_0_ex.item.method_printStackTrace;
               elseif ({JAVA_LANG_EXCEPTION} #?
                     current_thread_exception) /= Void then
                  try1_1_ex.item := {JAVA_LANG_EXCEPTION}
                        #? current_thread_exception;
                  try1_1_ex.item.method_printStackTrace;
               end
            end
            -- finally
            try1_catch := True; -- don't catch from finally
            (create { JAVA_LANG_SYSTEM_STATIC }).
                  field3_out_JAVA_LANG_SYSTEM.item.
                  method_println_from_string (create_java_string ("2"));
         rescue
            if not try1_catch then
               try1_catch := True;
               retry;
            end
         end).call ( [ local0_ex, local1_ex ] );
      (create { JAVA_LANG_SYSTEM_STATIC }).field3_out_JAVA_LANG_SYSTEM.
            item.method_println_from_string (create_java_string ("3"));
   end
```

All locals from the method are passed to the inline agent so that it can work with all variables. Since all variables are wrapped into a JAVA_VARIABLE, the references are the same and assignments done to them inside the agent will be visible in the method.
“current_thread_exception” is a once per thread variable and will be set upon throwing an exception (see next chapter).

4.36.33 ThrowStatement

Throwing an exception is done via the procedure “throw” from JAVA_PARENT:

```
throw new IllegalArgumentException("foobar");
```

becomes

```
throw ((create { JAVA_LANG_ILLEGALARGUMENTEXCEPTION }.
      make_from_string_JAVA_LANG_ILLEGALARGUMENTEXCEPTION (
      create_java_string ("foobar"))));
```

“throw” then saves the exception to “current_thread_exception” and triggers an Eiffel developer exception. This will abort the current routine and call its rescue block. There, the Java catch block (if present) will be executed by querying “current_thread_exception”.

4.36.34 SynchronizedStatement

In Java, every object has a monitor in order to implement synchronize, wait, notify and notifyAll. Static methods synchronize on the class instance. To support these operations in Eiffel, all classes become a monitor that overwrites the monitor of the parent class:

```
class
   TESTCLASS

inherit
   TESTCLASS_STATIC
   JAVA_LANG_OBJECT
      redefine
         mutex,
         monitor
      end

feature {ANY} -- mutex and monitor

   mutex: MUTEX is
      indexing
         once_status: global
      once
         create Result.make
      end

   monitor: CONDITION_VARIABLE is
      indexing
         once_status: global
      once
         create Result.make
      end

end
```

Synchronization can now be done on the mutex:

```
public synchronized void foo() {
   …
}
```

becomes

```
method_foo is
   do
      mutex.lock;
      …
      mutex.unlock;
   end
```

And synchronizing on an object

```
public void foo() {
   synchronized(obj) {
      …
   }
}
```

becomes

```
method_foo is
   do
      obj.mutex.lock;
      …
      obj.mutex.unlock;
   end
```

The “monitor” is used in “Object.wait()”, “Object.notify()” and “Object.notifyAll()”. It calls “monitor.wait (mutex)”, “monitor.signal” and “monitor.broadcast”.
This way, the Java monitor is fully implemented by using a MUTEX and a CONDITION_VARIABLE.

5 Java Runtime System

While complete support for all Java language constructs has been added, the runtime has only been implemented far enough to support basic SWT applications. But since most parts of the runtime system are at least partially implemented, getting a full fletched Java compiler only requires finishing programming all the runtime parts. No more basic research needs to be done.

5.1 Threading

Basic support for threading has been implemented by letting the created Eiffel code for the Java class java.lang.Thread additionally extend the Eiffel THREAD class. The native method “start0” (from java.lang.Thread) then calls the procedure “launch” (from THREAD; it starts the Eiffel thread).
java.lang.Thread is, because of this additional superclass, the only class that needs specific adaptions. All other OpenJDK classes can be compiled to Eiffel without any modifications.

5.2 Native libraries

All jvm libraries (e.g. java.dll) and third party libraries (e.g. swt-win32-XYZ.dll) except the JVM library jvm.dll can be used without modification with the created Eiffel code. jvm.dll is specific to the used OpenJDK and assumes its specific implementation of the Java runtime system. All other libraries are only able to interact with the Java (Eiffel) code via JNI. Since an own version of JNI has been implemented for the Java runtime in Eiffel (see below), the libraries can be kept and work out of the box as they are.
jvm.dll is therefore the only library that has been reimplemented to work with the conditions of the Java runtime for Eiffel. From those functions needed by a basic SWT application and the required OpenJDK classes, the ones that do not use VM specific features have been taken from the original implementation (e.g. JVM_CurrentTimeMillis, JVM_NanoTime). The ones where the original implementation does use specific OpenJDK VM features have been reimplemented (e.g. JVM_FindPrimitiveClass, JVM_ArrayCopy). The reimplemented functions have all been written in plain JNI without relying on the conditions of the Eiffel VM runtime. All other functions that are not needed by a basic SWT application have only been added as a stub printing the function name. So, whenever an unimplemented function is used (and the application probably crashes because the function is doing nothing), the implementation can be extended to also get the new application working.

5.3 Reflection

A basic decision made for this thesis is that all classes required at runtime are compiled into the binary. So there is no way of dynamically loading external classes at runtime. With this setup, it’s only necessary to being able get all information about classes already known at compile time. Since getting that exhaustive information at runtime is not possible in Eiffel (only basic attribute querying via INTERNAL and function invocation via CECIL exists), the information is added to all classes at compile time. For instance java.lang.String is getting this get_class query in the “static class” (JAVA_LANG_STRING_STATIC):

```
get_class : JAVA_CLASS is
   indexing
      once_status: global
   once
      create Result.make ("java.lang.String")
      Result.set_superclass (create {JAVA_LANG_OBJECT_STATIC});
      Result.add_field (create {JAVA_FIELD}.make (
         "CASE_INSENSITIVE_ORDER",
         "field0_CASE_INSENSITIVE_ORDER_JAVA_LANG_STRING", 25, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "count", "field1_count_JAVA_LANG_STRING", 18, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "hash", "field2_hash_JAVA_LANG_STRING", 2, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "offset", "field3_offset_JAVA_LANG_STRING", 18, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "serialPersistentFields",
         "field4_serialPersistentFields_JAVA_LANG_STRING", 26, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "serialVersionUID", "field5_serialVersionUID_JAVA_LANG_STRING",
         26, Result));
      Result.add_field (create {JAVA_FIELD}.make (
         "value", "field6_value_JAVA_LANG_STRING", 18, Result));
   end
```

JAVA_CLASS is a utility class that extends java.lang.Class and implements its methods based on the given data. Currently, only information about the superclass and basic information about all fields in the class are saved. It will be equally easy to also save information about all methods and annotations and then implement the corresponding java.lang.Class methods.
This is an expensive solution in terms of the required space (in the code and in compiled binaries), but it is relatively straightforward, simple and flexible.

5.4 Java Native Interface (JNI)

The Java developers have made an exceptional job while defining and implementing JNI because it is designed to be completely independent of the Java runtime implementation. In jni.h, the header file that is used to compile native libraries that implement native methods from Java classes, the JNI functions are only function pointers put together in a struct:

```
typedef const struct JNINativeInterface_ *JNIEnv;

struct JNINativeInterface_ {
    …
    jclass (JNICALL *FindClass)(JNIEnv *env, const char *name);
    …
};
```

With this JNI design, getting an own implementation is surprisingly easy and straightforward. A “JNIEnv” struct is set up and all function pointers are set to own functions implementing the required functionality with CECIL. This JNIEnv struct is then created once at runtime and given to all DLL calls in the Eiffel code (as seen in MethodInvocation chapter (4.36.10)). Access to the JNIEnv is done in JAVA_PARENT:

```
jni_env: POINTER is
   indexing
      once_status: global
   once
      Result := jni_init
   end

jni_init: POINTER is
   external
      "C"
   end
```

“jni_init” is written in C and the compiled object is added to the Eiffel compilation at linking time. So the JNI implementation is not in an own library (dll) but in the final compiled binary instead.
Implementing the functionality of the different JNI functions is rather easy with CECIL because the functions provide quite exactly the same kind of access to Java classes as CECIL does for Eiffel classes (after all, the basic idea behind these two technologies is the same). The interested reader is invited to take a look at the file “jnienv.c” to see how the mapping from JNI to CECIL is done.

6 Open issues

In this chapter, some open issues, including simplifications that have been done while implementing the runtime, are discussed.

6.1 String encoding

The implementation of JNI assumes all strings to be plain ASCII. While this assumption is ok for simple SWT applications, it of course does not hold for real world application. So the whole JNI implementation needs to support the required Unicode and UTF8 character operations. This should be quite easy since it can mostly be taken from the original JNI implementation.

6.2 Garbage Collection

JNI and CECIL both need to properly protect objects that have been created in the native code to not have the garbage collector deleting them prematurely. This is not yet done in the current JNI implementation and therefore, garbage collection is currently being turned off.
This is of course ok to get an experimental implementation of the Java runtime in Eiffel but needs to be fixed for real world applications. Implementing proper protection should be quite easy since there are specific JNI functions where the protection and protection removing can be done. Since CECIL offers quite exactly the same procedures, it can easily be mapped.

6.3 Object finalization

Finalizing objects has not been implemented since it’s not important for basic SWT applications and because the garbage collector is turned off anyway. Implementing finalization should be quite easy by letting JAVA_PARENT extend MEMORY and by redefining the dispose procedure so that it calls Object.finalize().

6.4 Serialization

I have not looked into serialization in my thesis but since there are serialization mechanisms in Eiffel, it would at least be possible to implement an own version of

serialization (the mechanism of Eiffel needs to be adapted to omit transient fields (can be identified via reflection)). But doing an own implementation would make the saved data proprietary and not compatible to Java. Therefore, two processes where one is compiled via Eiffel and one running on a Sun JVM couldn't communicate with each other if the data is exchanged via serialization. Maybe it is possible to use the Java way of serialization. This has to be evaluated.

6.5 Soft, Weak and Phantom References

I have not looked into this topic closely but from the sourcecode of the three classes, it appears that the intended reference behaviour is achieved by using a daemon thread. Since there are not native methods involved and all Java language constructs have been mapped, this should work by default.

6.6 Binary size

The binaries compiled from the created Eiffel code are quite big (~36mb for a simple HelloWorld).
The major reason for this is that in Java, classes are highly interconnected. For instance a simple HelloWorld application uses java.lang.System (for System.out.println("HelloWorld")), System then uses java.lang.SecuritManager which uses java.awt.AWTPermission and this way, a simple HelloWorld already pulls in big parts of the AWT and Swing code.
One way to work around this specific problem is to remove whole packages from the compilation and adapt the remaining classes to not use that code. For instance a HelloWorld doesn't need the java.awt package. So, all classes from that package could be excluded from the compilation and code referencing it could be rewritten. A method for instance using one of these classes could be replaced by a method doing nothing. A similar technique is used by JNC [3], a Java to native compiler based on GCJ [2], that I wrote about a year ago. A simple HelloWorld will result in a ~21mb binary with JNC. When excluding GUI relevant packages (java.applet, java.awt, java.text, javax.imageio, javax.print and javax.swing), the binary will only be 8mb and still work flawlessly because the excluded code is never used at runtime. Of course an application will crash if the excluded code will actually being executed at some point (for instance a method that is doing nothing won't probably create a desired result or effect).
This idea of excluding whole packages could be implemented very easily because it only requires a very small adaption in the code that is doing the dependency search (searching all required classes of the source provided from the user).

Other minor reasons for big binaries are:

- All methods of the compiled code are set to being visible because JNI might need to access them. This makes dead code removal useless (therefore it's disabled by default). By manually enabling dead code removal and only adding required features as visible, the binary would become smaller. A HelloWorld application for instance results in a ~27mb binary instead of ~36mb this way.
- The extensive reflection data has the same issue. It is added for all classes but most likely used for only very few of them. Adding a mechanism that allows to only adding the data for selected classes would also decrease the binary size.
- The usage of JAVA_VARIABLE in methods is only required in specific circumstances (for instance if a try-block is present or new values are assigned to method parameters) but currently it is always used. By adding another analysis step to identify cases where the JAVA_VARIABLE is needed and not using it in

the other cases, it would reduce the size of the binary as well. Worthwhile mentioning is that for fields, the wrapping into JAVA_VARIABLE is always needed because this allows assigning values to the fields without needing to create setter methods.
- One last point that is also used in JNC is binary packing. By using a binary packer (UPX [4] for instance), the size can be reduced to about 25% to 30% of the original size. But of course this only moves the problem to startup speed and memory consumption since the packed binary will have do decompress the original binary into the memory on startup and then run it.

The most promising approach is definitely the automatic exclusion based on unneeded packages.

7 References

[1] Excelsior JET, http://www.excelsior-usa.com/jet.html
[2] GCJ - The Gnu Compiler for the Java Programming Language, http://gcc.gnu.org/java/
[3] JNC - JavaNativeCompiler, http://jnc.mtsystems.ch/
[4] UPX - the Ultimate Packer for eXecutables, http://upx.sourceforge.net/

The created compiler, its source code as well as all mentioned files in this documentation can be found on http://jaftec.origo.ethz.ch/.