

## CSc 620

# Debugging, Profiling, Tracing, and Visualizing Programs

## 2 : Java Bytecode — BCEL

Christian Collberg

collberg+620@gmail.com

Department of Computer Science  
University of Arizona

Copyright © 2005 Christian Collberg

—Fall 2005 — 2

[1]

- BCEL (formerly `JavaClass`) allows you to load a class, iterate through the methods and fields, change methods, add new methods and fields, etc.
- BCEL also makes it easy to create new classes from scratch.

620 —Fall 2005 — 2

[2]

## Getting started

To open a class file for editing you do the following:

```
org.apache.bcel.classfile.ClassParser p =  
    new org.apache.bcel.classfile.ClassParser(classFile);  
org.apache.bcel.classfile.JavaClass jc = p.parse()  
org.apache.bcel.generic.ClassGen cg =  
    new org.apache.bcel.generic.ClassGen(jc);  
org.apache.bcel.generic.ConstantPoolGen cp =  
    new org.apache.bcel.generic.ConstantPoolGen(  
        jc.getConstantPool());
```

Fall 2005 — 2

[3]

## Getting started...

- A `org.apache.bcel.classfile.JavaClass`-object represents the parsed class file. You can query this object to get any information you need about the class, for example its methods, fields, code, super-class, etc.
- A `org.apache.bcel.generic.ClassGen`-object represents a class that can be edited. You can add methods and fields, you can modify existing methods, etc.
- A `org.apache.bcel.generic.ConstantPoolGen`-object represents a constant pool to which new constants can be added.

620 —Fall 2005 — 2

[4]

## Finishing up

- When you are finished editing the class you must close and save it to the new class file:

```
org.apache.bcel.classfile.JavaClass jcl =
    cg.getJavaClass();
jcl.setConstantPool(cp.getFinalConstantPool());
jcl.dump(classFile1);
```

## Editing a Method

- `org.apache.bcel.classfile.Method` represents a method that has been read from a class file.
- `org.apache.bcel.generic.MethodGen` is the BCEL class that represents a method being edited:

```
org.apache.bcel.generic.ConstantPoolGen cp = ...;
org.apache.bcel.generic.ClassGen cg = ...;

org.apache.bcel.classfile.Method method =
    cg.containsMethod(methodName, methodSignature);
org.apache.bcel.generic.MethodGen mg =
    new org.apache.bcel.generic.MethodGen(method, cname, cp);
```

## Editing Instructions

- Once you have opened a method for editing you can get its list of instructions:

```
org.apache.bcel.generic.MethodGen mg = ...;

org.apache.bcel.generic.InstructionList il =
    mg.getInstructionList();
org.apache.bcel.generic.InstructionHandle[] ihs =
    il.getInstructionHandles();
for(int i=0; i < ihs.length; i++) {
    org.apache.bcel.generic.InstructionHandle ih = ihs[i];
    org.apache.bcel.generic.Instruction instr =
        ih.getInstruction();
}
```

## Editing Instructions...

- `org.apache.bcel.generic.InstructionLists` can be manipulated by appending or inserting new instruction, deleting instructions, etc.
- All bytecode instructions in BCEL are represented by their own class. `org.apache.bcel.generic.IADD` is the class that represents integer addition, for example.
- This allows us to use `instanceof` to classify instructions.

# Changing an Instruction

```
org.apache.bcel.generic.Instruction inst = ...;
org.apache.bcel.generic.InstructionHandle ih = ...;

if (inst instanceof org.apache.bcel.generic.INVOKESTATIC){
    org.apache.bcel.generic.INVOKESTATIC call =
        (org.apache.bcel.generic.INVOKESTATIC) inst;
    String className = call.getClassName(cp);
    String methodName = call.getMethodName(cp);
    String methodSig = call.getSignature(cp);

    ih.setInstruction(new org.apache.bcel.generic.NOP());
}
```

- `setInstruction` replaces an instruction with a new one.

# Local variables

- A Java method can have a maximum of 256 local variables. (Actually 65536).
- In a virtual (non-static) method local variable zero (“slot #0”) is `this`.
- Formal arguments take up the first slots. In other words, in a virtual method with 2 formal parameters, slot #0 is `this`, slot #1 is the first formal, slot #2 is the second formal, and any local variables are slots #3,#4,...,#255.

## Local variables...

- Local variables can be reused within a method.
- A clever compiler will allocate both `x` and `y` to the same slot, since they have non-overlapping lifetimes.
- For example, consider the following method:

```
void P() {
    int x= 5;
    System.out.println(x);
    float y=5.6;
    System.out.println(y);
}
```

## Local variables...

- BCEL requires that we indicate where a new local variable is accessible:

```
org.apache.bcel.generic.Type T = ...;

org.apache.bcel.generic.LocalVariableGen lg =
    mg.addLocalVariable(localName, T, null, null);
int localIndex = lg.getIndex();

// push something here
org.apache.bcel.generic.Instruction store =
    new org.apache.bcel.generic.ASTORE(localIndex);

org.apache.bcel.generic.InstructionHandle start = ...;
lg.setStart(start);
```

## Local variables...

- `org.apache.bcel.generic.LocalVariableGen` creates a new local variable.
- We start these locations out as `null` (unknown) and fill them in using the `setStart` method.
- It takes an `org.apache.bcel.generic.InstructionHandle` as input, namely the first instruction where the variable should be visible.

```
org.apache.bcel.generic.LocalVariableGen lg =
    mg.addLocalVariable(localName, T, null, null);
...
org.apache.bcel.generic.InstructionHandle start = ...;
lg.setStart(start);
```

## Example 1: Creating a New Class

- We create a new `org.apache.bcel.generic.ClassGen` object representing the class.

```
int flags = org.apache.bcel.Constants.ACC_PUBLIC;
String class_name = "MyClass";
String file_name = "MyClass.java";
String super_class_name = "java.lang.Object";
String[] interfaces = {};

org.apache.bcel.generic.ClassGen cg =
    new org.apache.bcel.generic.ClassGen(
        class_name, super_class_name, file_name,
        flags, interfaces);
```

## Example 1...

- We can then add a field `field1` of type `int` to the new class:

```
org.apache.bcel.generic.ConstantPoolGen cp =
    cg.getConstantPool();

org.apache.bcel.generic.Type field_type =
    org.apache.bcel.generic.Type.INT;
String field_name = "field1";

org.apache.bcel.generic.FieldGen fg =
    new org.apache.bcel.generic.FieldGen(
        access_flags, field_type, field_name, cp);
cg.addField(fg.getField());
```

## Example 1 – Creating a Method

```
int method_access_flags =
    org.apache.bcel.Constants.ACC_PUBLIC |
    org.apache.bcel.Constants.ACC_STATIC;

org.apache.bcel.generic.Type return_type =
    org.apache.bcel.generic.Type.VOID;

org.apache.bcel.generic.Type[] arg_types =
    org.apache.bcel.generic.Type.NO_ARGS;
String[] arg_names = {};
```

## Example 1 – Creating a Method...

```
String method_name = "method1";
String class_name = "MyClass";

org.apache.bcel.generic.InstructionList il =
    new org.apache.bcel.generic.InstructionList();
il.append(
    org.apache.bcel.generic.InstructionConstants.RETURN);

org.apache.bcel.generic.MethodGen mg =
    new org.apache.bcel.generic.MethodGen(
        method_access_flags, return_type, arg_types,
        arg_names, method_name, class_name, il, cp);

mg.setMaxStack();
cg.addMethod(mg.getMethod());
```

—Fall 2005 — 2

[17]

## Branches

- As is always the case with branches, we need to be able to handle forward jumps.
- The typical way of accomplishing this is to create a branch with unspecified target:

```
org.apache.bcel.generic.InstructionList il = ...;

org.apache.bcel.generic.IFNULL branch =
    new org.apache.bcel.generic.IFNULL(null);
il.append(branch);
...
```

Fall 2005 — 2

[19]

## Example 1 – Creating a Method...

- The method has no arguments, returns `void`, and contains only one instruction, a `return`.
- Note the call to `mg.setMaxStack()`. With every method in a class file is stored the maximum stack-size is needed to execute the method.
- For example, a method whose body consists of the instructions `{iconst_1, iconst_2, iadd, pop}` (i.e. compute  $1 + 2$  and throw away the result) will have `max-stack` set to two.
- We can either compute this ourselves or let BCEL's `mg.setMaxStack()` do it for us.

620 —Fall 2005 — 2

[18]

## Branches...

- When we have gotten to the location we want to jump to we add a (bogus) `NOP` instruction which will serve our branch target.
- The `append` instruction returns a handle to the `NOP` and we use this handle to set the target of the branch:

```
org.apache.bcel.generic.InstructionHandle h =
    il.append(new org.apache.bcel.generic.NOP());
branch.setTarget(h);
```

620 —Fall 2005 — 2

[20]

# Exceptions

Here is the generic code for building a try-catch-block:

```
org.apache.bcel.generic.InstructionList il = ...;
org.apache.bcel.generic.MethodGen mg = ...;
String exception = "java.lang.Exception";

org.apache.bcel.generic.InstructionHandle start_pc =
    il.append(new org.apache.bcel.generic.NOP());

// The code that builds the try-body goes here.

// Code to jump out of the try-block:
org.apache.bcel.generic.GOTO branch =
    new org.apache.bcel.generic.GOTO(null);
```

# Exceptions...

```
org.apache.bcel.generic.InstructionHandle end_pc =
    il.append(branch);

// Pop exception off stack when entering catch block.
org.apache.bcel.generic.InstructionHandle handler_pc =
    il.append(new org.apache.bcel.generic.POP());

// The code that builds the catch-body goes here.

// Add a NOP after the exception handler. This is
// where we will jump after we're through with the
// try-block.
org.apache.bcel.generic.InstructionHandle next_pc =
    il.append(new org.apache.bcel.generic.NOP());
branch.setTarget(next_pc);
```

# Exceptions...

```
org.apache.bcel.generic.ObjectType catch_type =
    new org.apache.bcel.generic.ObjectType(exception);

org.apache.bcel.generic.CodeExceptionGen eg =
    mg.addExceptionHandler(
        start_pc, end_pc, handler_pc, catch_type);
```

start\_pc and end\_pc hold the beginning and the end of the try-body, respectively. handler\_pc holds the address of the beginning of the catch-block.

# Types

BaseType	Type	Interpretation
<b>B</b>	byte	signed 8-bit integer
<b>C</b>	char	Unicode character
<b>D</b>	double	64-bit floating point value
<b>F</b>	float	32-bit floating point value
<b>I</b>	int	32-bit integer
<b>J</b>	long	64-bit integer
<b>L&lt;classname&gt;;</b>	reference	instance of class <classname>
<b>S</b>	short	signed 16-bit integer
<b>Z</b>	boolean	true or false
<b>[</b>	reference	one array dimension
<b>V</b>		void
<b>(BaseType*)BaseType</b>		method descriptor

## Types...

- A source-level Java type is encoded into a classfile type descriptor using the definitions on the previous slide.

- Types (such as method signatures) are defined by the class `org.apache.bcel.generic.Type`:

```
org.apache.bcel.generic.Type return_type =
    org.apache.bcel.generic.Type.VOID;
org.apache.bcel.generic.Type[] arg_types =
    new org.apache.bcel.generic.Type[] {
        new org.apache.bcel.generic.ArrayType(
            org.apache.bcel.generic.Type.STRING, 1)
    };
```

## Types...

- BCEL has quite a number of methods that convert back and forth between Java source format

```
java.lang.Object[]
```

bytecode format

```
[Ljava/lang/Object;
```

and BCEL's internal format `org.apache.bcel.generic.Type`.

- `org.apache.bcel.generic.Type.getType` converts from Java bytecode format to Java source format:

```
String S = "[Ljava/lang/Object;";
org.apache.bcel.generic.Type T =
    org.apache.bcel.generic.Type.getType(S);
System.out.println(T); // ==> "java.lang.Object[]".
```

## Types...

- The method `org.apache.bcel.classfile.Utility.getSignature` converts a type in Java bytecode format to Java source format:

```
String type = "java.lang.Object[]";
String S =
    org.apache.bcel.classfile.Utility.getSignature(type);
System.out.println(S); ==> "[Ljava/lang/Object;"
```

## Types...

- The methods `org.apache.bcel.generic.Type.getArgumentTypes` and `org.apache.bcel.generic.Type.getReturnType` take a type in Java bytecode format as argument and extract the array of argument types and the return type, respectively:

- `org.apache.bcel.generic.Type.getMethodSignature` converts the return and argument types back to a Java bytecode type string.

## Types...

```
String S = "(Ljava/lang/String;I)V";
org.apache.bcel.generic.Type[] arg_types =
    org.apache.bcel.generic.Type.getArgumentTypes(S);
org.apache.bcel.generic.Type return_type =
    org.apache.bcel.generic.Type.getReturnType(S);
String M =
    org.apache.bcel.generic.Type.getMethodSignature(
        return_type, arg_types);
```

## Types...

- The method `org.apache.bcel.generic.Type.getSignature` converts a BCEL type to the equivalent Java bytecode signature. This code

```
org.apache.bcel.generic.Type T =
    org.apache.bcel.generic.Type.STRINGBUFFER;
String M = T.getSignature();
System.out.println(M);
```

will print out `Ljava/lang/StringBuffer;`.

## Static Method Calls

- To make a static method call we push the arguments on the stack, create a constant pool reference to the method, and then generate an `INVOKESTATIC` opcode that performs the actual call:

```
String className = ...;
String methodName = ...;
String signature = ...;

org.apache.bcel.generic.InstructionList il = ...;
org.apache.bcel.generic.ConstantPoolGen cp = ...;
```

## Static Method Calls...

```
// Generate code that pushes the actual
// arguments of the call.

int index =
    cp.addMethodref(className, methodName, signature);
org.apache.bcel.generic.INVOKESTATIC call =
    new org.apache.bcel.generic.INVOKESTATIC(index);
il.append(call);
```



## Virtual Method Calls

- Making a virtual call is similar, except that we need an object to make the call through. This object reference is pushed on the stack prior to the arguments:

```
// Generate code pushing the object on the stack.
// Generate code pushing the actual arguments.

int index =
    cp.addMethodref(className,methodName,signature);
org.apache.bcel.generic.INVOKEVIRTUAL s =
    new org.apache.bcel.generic.INVOKEVIRTUAL(index);
il.append(call);
```

## Example 2: List.java

```
public class List {
public static void main(String args[])
    throws java.io.IOException {
    String classFile = args[0];
    String className =
        classFile.substring(
            0,classFile.length()-".class".length());

    org.apache.bcel.classfile.ClassParser p =
        new org.apache.bcel.classfile.ClassParser(classFile);

    org.apache.bcel.classfile.JavaClass jc = p.parse();

    org.apache.bcel.generic.ClassGen cg =
        new org.apache.bcel.generic.ClassGen(jc);
```

## Example 2: List.java

```
org.apache.bcel.classfile.ConstantPool cp =
    jc.getConstantPool();

org.apache.bcel.generic.ConstantPoolGen cpg =
    new org.apache.bcel.generic.ConstantPoolGen(cp);

org.apache.bcel.classfile.Method[] methods =
    cg.getMethods();

for(int m=0; m<methods.length; m++) {
    org.apache.bcel.generic.MethodGen mg =
        new org.apache.bcel.generic.MethodGen(
            methods[m],className,cpg);
    System.out.println("\nMETHOD: " +
        mg.getClassName() + "." +
        mg.getName() + ":" + mg.getSignature());
```

## Example 2: List.java

```
org.apache.bcel.generic.InstructionList il =
    mg.getInstructionList();
org.apache.bcel.generic.InstructionHandle[] ihs =
    il.getInstructionHandles();

int pc = 0;
for(int i=0; i < ihs.length; i++) {
    org.apache.bcel.generic.InstructionHandle ih = ihs[i];
    org.apache.bcel.generic.Instruction instr =
        ih.getInstruction();
    System.out.println(pc + " : " + instr.toString(cp));
    pc += instr.getLength();
}
}
```

# Readings and References

---

● BCEL is here: <http://jakarta.apache.org/bcel/index.html>. There is a manual and an on-line API description at <http://bcel.sourceforge.net/docs/index.html>.