University of Arizona

CSc 620

Security Through Obscurity

Christian Collberg

January 23, 2002

BCEL

BCEL

- `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.

Slide 3–1

Getting started

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

```
de.fub.bytecode.classfile.ClassParser p =
   new de.fub.bytecode.classfile.ClassParser(classFile);
 de.fub.bytecode.classfile.JavaClass jc = p.parse()
de.fub.bytecode.generic.ClassGen cg =
   new de.fub.bytecode.generic.ClassGen(jc);
de.fub.bytecode.generic.ConstantPoolGen cp =
   new de.fub.bytecode.generic.ConstantPoolGen(
              jc.getConstantPool());
```

Slide 3–2

Getting started...

- A `de.fub.bytecode.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 `de.fub.bytecode.generic.ClassGen`-object represents a class that can be edited. You can add methods and fields, you can modify existing methods, etc.
- A `de.fub.bytecode.generic.ConstantPoolGen`-object represents a constant pool to which new constants can be added.

Slide 3–3

Finishing up

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

```
de.fub.bytecode.classfile.JavaClass jc1 =
   cg.getJavaClass();
jc1.setConstantPool(cp.getFinalConstantPool());
jc1.dump(classFile1);
```


Editing a Method

- `de.fub.bytecode.classfile.Method` represents a method that has been read from a class file.
- `de.fub.bytecode.generic.MethodGen` is the `BCEL` class that represents a method being edited:

```
de.fub.bytecode.generic.ConstantPoolGen cp = ...;
de.fub.bytecode.generic.ClassGen cg = ...;

de.fub.bytecode.classfile.Method method =
   cg.containsMethod(methodName, methodSignature);
de.fub.bytecode.generic.MethodGen mg =
   new de.fub.bytecode.generic.MethodGen(method,cname,cp);
```


Editing Instructions

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

```
de.fub.bytecode.generic.MethodGen mg = ...;

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


Editing Instructions...

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

Slide 3–8

Changing an Instruction

```
de.fub.bytecode.generic.Instruction inst = ...;
de.fub.bytecode.generic.InstructionHandle ih = ...;

if (inst instanceof de.fub.bytecode.generic.INVOKESTATIC){
   de.fub.bytecode.generic.INVOKESTATIC call =
     (de.fub.bytecode.generic.INVOKESTATIC) inst;
   String className  = call.getClassName(ec.cp);
   String methodName = call.getMethodName(ec.cp);
   String methodSig  = call.getSignature(ec.cp);

   ih.setInstruction(new de.fub.bytecode.generic.NOP());
}
```

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

Slide 3–9

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.

Slide 3–10

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);
}
```

Slide 3–11

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

```
de.fub.bytecode.generic.Type T = ...;

de.fub.bytecode.generic.LocalVariableGen lg =
   mg.addLocalVariable(localName, T, null, null);
int localIndex = lg.getIndex();

// push something here
de.fub.bytecode.generic.Instruction store =
   new de.fub.bytecode.generic.ASTORE(localIndex);

de.fub.bytecode.generic.InstructionHandle start = ...;
lg.setStart(start);
```

Slide 3–12

Local variables...

- `de.fub.bytecode.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 `de.fub.bytecode.generic.InstructionHandle` as input, namely the first instruction where the variable should be visible.

```
de.fub.bytecode.generic.LocalVariableGen lg =
   mg.addLocalVariable(localName, T, null, null);
...
de.fub.bytecode.generic.InstructionHandle start = ...;
lg.setStart(start);
```

Slide 3–13

Example 1: Creating a New Class

- We create a new `de.fub.bytecode.generic.ClassGen` object representing the class.

```
int flags = de.fub.bytecode.Constants.ACC_PUBLIC;
String class_name = "MyClass";
String file_name = "MyClass.java";
String super_class_name = "java.lang.Object";
String[] interfaces = {};

de.fub.bytecode.generic.ClassGen cg =
   new de.fub.bytecode.generic.ClassGen(
      class_name, super_class_name, file_name,
      flags, interfaces);
```

Slide 3–14

Example 1...

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

```
de.fub.bytecode.generic.ConstantPoolGen cp =
   cg.getConstantPool();

de.fub.bytecode.generic.Type field_type =
   de.fub.bytecode.generic.Type.INT;
String field_name = "field1";

de.fub.bytecode.generic.FieldGen fg =
   new de.fub.bytecode.generic.FieldGen(
      access_flags, field_type, field_name, cp);
cg.addField(fg.getField());
```

Slide 3–15

Example 1 – Creating a Method

```
int method_access_flags =
   de.fub.bytecode.Constants.ACC_PUBLIC |
   de.fub.bytecode.Constants.ACC_STATIC;

de.fub.bytecode.generic.Type return_type =
   de.fub.bytecode.generic.Type.VOID;

de.fub.bytecode.generic.Type[] arg_types =
   de.fub.bytecode.generic.Type.NO_ARGS;
String[] arg_names = {};
```

Slide 3–16

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

de.fub.bytecode.generic.InstructionList il =
   new de.fub.bytecode.generic.InstructionList();
il.append(
   de.fub.bytecode.generic.InstructionConstants.RETURN);

de.fub.bytecode.generic.MethodGen mg =
   new de.fub.bytecode.generic.MethodGen(
      method_access_flags, return_type,  arg_types,
      arg_names, method_name, class_name, il, cp);

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

Slide 3–17

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.

Slide 3–18

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:

```
de.fub.bytecode.generic.InstructionList il = ...;

de.fub.bytecode.generic.IFNULL branch =
   new de.fub.bytecode.generic.IFNULL(null);
il.append(branch);
   ...
```

Slide 3–19

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:

```
de.fub.bytecode.generic.InstructionHandle h =
   il.append(new de.fub.bytecode.generic.NOP());
branch.setTarget(h);
```

Slide 3–20

Exceptions

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

```
de.fub.bytecode.generic.InstructionList il = ...;
de.fub.bytecode.generic.MethodGen mg = ...;
String exception = "java.lang.Exception";

de.fub.bytecode.generic.InstructionHandle start_pc =
   il.append(new de.fub.bytecode.generic.NOP());

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

// Code to jump out of the try-block:
de.fub.bytecode.generic.GOTO branch =
   new de.fub.bytecode.generic.GOTO(null);
```

Slide 3–21

```
de.fub.bytecode.generic.InstructionHandle end_pc =
   il.append(branch);

// Pop exception off stack when entering catch block.
de.fub.bytecode.generic.InstructionHandle handler_pc =
    il.append(new de.fub.bytecode.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.
de.fub.bytecode.generic.InstructionHandle next_pc =
    il.append(new de.fub.bytecode.generic.NOP());
branch.setTarget(next_pc);
```

Slide 3–22

```
de.fub.bytecode.generic.ObjectType catch_type =
   new de.fub.bytecode.generic.ObjectType(exception);

de.fub.bytecode.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.

Slide 3–23

BaseType	Type	Interpretation
B	byte	signed 8-bit integer
C	char	Unicode character
D	double	64-bit floating point value
F	float	32-bit floating point value
I	int	32-bit integer
J	long	64-bit integer
L<*classname*>;	reference	instance of class <*classname*>
S	short	signed 16-bit integer
Z	boolean	`true` pr `false`
[	reference	one array dimension
V		void
(BaseType*)BaseType		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 `de.fub.bytecode.generic.Type`:

```
de.fub.bytecode.generic.Type return_type =
   de.fub.bytecode.generic.Type.VOID;
de.fub.bytecode.generic.Type[] arg_types   =
   new de.fub.bytecode.generic.Type[] {
      new de.fub.bytecode.generic.ArrayType(
         de.fub.bytecode.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 `de.fub.bytecode.generic.Type`.
- `de.fub.bytecode.generic.Type.getType` converts from Java bytecode format to Java source format:

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


Types. . .

- The method `de.fub.bytecode.classfile.Utility.getSignature` converts a type in Java bytecode format to Java source format:

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


Types. . .

- The methods `de.fub.bytecode.generic.Type.getArgumentTypes` and `de.fub.bytecode.generic.Type.getReturnType` take a type in Java bytecode format as argument and extract the array of argument types and the return type, respectively:
- `de.fub.bytecode.generic.Type.getMethodSignature` converts the return and argument types back to a Java bytecode type string.

Slide 3–28

Types. . .

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

Slide 3–29

Types. . .

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

  ```
  de.fub.bytecode.generic.Type T =
     de.fub.bytecode.generic.Type.STRINGBUFFER;
  String M = T.getSignature();
  System.out.println(M);
  ```

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

Slide 3–30

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  = ...;

de.fub.bytecode.generic.InstructionList il = ...;
de.fub.bytecode.generic.ConstantPoolGen cp = ...;
```

Slide 3–31

Static Method Calls. . .

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

int index =
   cp.addMethodref(className,methodName,signature);
de.fub.bytecode.generic.INVOKESTATIC call =
    new de.fub.bytecode.generic.INVOKESTATIC(index);
il.append(call);
```

Slide 3–32

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);
de.fub.bytecode.generic.INVOKEVIRTUAL s =
    new de.fub.bytecode.generic.INVOKEVIRTUAL(index);
il.append(call);
```

Slide 3–33

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());

   de.fub.bytecode.classfile.ClassParser p =
      new de.fub.bytecode.classfile.ClassParser(classFile);

   de.fub.bytecode.classfile.JavaClass jc = p.parse();

   de.fub.bytecode.generic.ClassGen cg =
      new de.fub.bytecode.generic.ClassGen(jc);
```

Slide 3–34

```
   de.fub.bytecode.classfile.ConstantPool cp =
       jc.getConstantPool();

   de.fub.bytecode.generic.ConstantPoolGen cpg =
      new de.fub.bytecode.generic.ConstantPoolGen(cp);

   de.fub.bytecode.classfile.Method[] methods =
      cg.getMethods();

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

Slide 3–35

```
      de.fub.bytecode.generic.InstructionList il =
         mg.getInstructionList();
      de.fub.bytecode.generic.InstructionHandle[] ihs =
         il.getInstructionHandles();

      int pc = 0;
      for(int i=0; i < ihs.length; i++) {
         de.fub.bytecode.generic.InstructionHandle ih = ihs[i];
         de.fub.bytecode.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`.