```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import java.io.EOFException;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.InputStream;
import java.util.ArrayList;
import static jminusminus.CLConstants.*;

/**
 * CLAbsorber is for reading a Java class into an in-memory CLFile
 * representation and printing it out to STDOUT in a format similar to that of
 * javap.
 */

public class CLAbsorber {

    /** CLFile representation of the class that is read. */
    private CLFile classFile;

    /** Whether or not an error occurred in reading the class. */
    private boolean errorHasOccurred;

    /** Name of the class that is read. */
    private String className;

    /**
     * Print the specified warning to STDERR.
     *
     * @param message
     *            warning.
     * @param args
     *            related values.
     */

    private void reportWarning(String message, Object... args) {
        System.err.printf("CLAbsorber Warning: " + message + "\n", args);
    }

    /**
     * Print the specified error message to STDERR and set the error flag to
     * true.
     *
     * @param message
     *            error message.
     * @param args
     *            related values.
     */

    private void reportError(String message, Object... args) {
        System.err.printf("CLAbsorber Error: " + message + "\n", args);
        errorHasOccurred = true;
    }

    /**
     * Read the constant pool information from the specified stream, and return
     * the information as a CLConstantPool object.
     *
     * @param in
     *            input stream.
     * @return the constant pool.
     */

    private CLConstantPool readConstantPool(CLInputStream in) {
```

```
        CLConstantPool cp = new CLConstantPool();
        try {
            for (int i = 1; i < classFile.constantPoolCount; i++) {
                int tag = in.readUnsignedByte();
                switch (tag) {
                case CONSTANT_Class:
                    cp
                            .addCPItem(new CLConstantClassInfo(in
                                    .readUnsignedShort()));
                    break;
                case CONSTANT_Fieldref:
                    cp.addCPItem(new CLConstantFieldRefInfo(in
                            .readUnsignedShort(), in.readUnsignedShort()));
                    break;
                case CONSTANT_Methodref:
                    cp.addCPItem(new CLConstantMethodRefInfo(in
                            .readUnsignedShort(), in.readUnsignedShort()));
                    break;
                case CONSTANT_InterfaceMethodref:
                    cp.addCPItem(new CLConstantInterfaceMethodRefInfo(in
                            .readUnsignedShort(), in.readUnsignedShort()));
                    break;
                case CONSTANT_String:
                    cp.addCPItem(new CLConstantStringInfo(in
                            .readUnsignedShort()));
                    break;
                case CONSTANT_Integer:
                    cp.addCPItem(new CLConstantIntegerInfo(in.readInt()));
                    break;
                case CONSTANT_Float:
                    cp.addCPItem(new CLConstantFloatInfo(in.readFloat()));
                    break;
                case CONSTANT_Long:
                    cp.addCPItem(new CLConstantLongInfo(in.readLong()));
                    i++;
                    break;
                case CONSTANT_Double:
                    cp.addCPItem(new CLConstantDoubleInfo(in.readDouble()));
                    i++;
                    break;
                case CONSTANT_NameAndType:
                    cp.addCPItem(new CLConstantNameAndTypeInfo(in
                            .readUnsignedShort(), in.readUnsignedShort()));
                    break;
                case CONSTANT_Utf8:
                    int length = in.readUnsignedShort();
                    byte[] b = new byte[length];
                    in.read(b);
                    cp.addCPItem(new CLConstantUtf8Info(b));
                    break;
                default:
                    reportError("Unknown cp_info tag '%d'", tag);
                    return cp;
                }
            }
        } catch (IOException e) {
            reportError("Error reading constant pool from file %s", className);
        }
        return cp;
    }

    /**
     * Read the fields from the specified stream, and return them as a list.
     *
     * @param in
     *            input stream.
     * @param fieldsCount
     *            number of fields.
     * @return list of fields.
```

```
     */

    private ArrayList<CLFieldInfo> readFields(CLInputStream in, int fieldsCount)
{
        ArrayList<CLFieldInfo> fields = new ArrayList<CLFieldInfo>();
        try {
            for (int i = 0; i < fieldsCount; i++) {
                int accessFlags = in.readUnsignedShort();
                int nameIndex = in.readUnsignedShort();
                int descriptorIndex = in.readUnsignedShort();
                int attributesCount = in.readUnsignedShort();
                fields.add(new CLFieldInfo(accessFlags, nameIndex,
                        descriptorIndex, attributesCount, readAttributes(in,
                                attributesCount)));
            }
        } catch (IOException e) {
            reportError("Error reading fields from file %s", className);
        }
        return fields;
    }

    /**
     * Read the methods from the specified stream, and return them as a list.
     *
     * @param in
     *            input stream.
     * @param methodsCount
     *            number of methods.
     * @return the methods.
     */

    private ArrayList<CLMethodInfo> readMethods(CLInputStream in,
            int methodsCount) {
        ArrayList<CLMethodInfo> methods = new ArrayList<CLMethodInfo>();
        try {
            for (int i = 0; i < methodsCount; i++) {
                int accessFlags = in.readUnsignedShort();
                int nameIndex = in.readUnsignedShort();
                int descriptorIndex = in.readUnsignedShort();
                int attributesCount = in.readUnsignedShort();
                methods.add(new CLMethodInfo(accessFlags, nameIndex,
                        descriptorIndex, attributesCount, readAttributes(in,
                                attributesCount)));
            }
        } catch (IOException e) {
            reportError("Error reading methods from file %s", className);
        }
        return methods;
    }

    /**
     * Read the attributes from the specified stream, and return them as a list
     *
     * @param in
     *            input stream.
     * @param attributeCount
     *            number of attributes.
     * @return list of attributes.
     */

    private ArrayList<CLAttributeInfo> readAttributes(CLInputStream in,
            int attributesCount) {
        ArrayList<CLAttributeInfo> attributes = new ArrayList<CLAttributeInfo>();
        try {
            CLConstantPool cp = classFile.constantPool;
            for (int i = 0; i < attributesCount; i++) {
                int attributeNameIndex = in.readUnsignedShort();
                long attributeLength = in.readUnsignedInt();
                CLAttributeInfo attributeInfo = null;
```

```
                String attributeName = new String(((CLConstantUtf8Info) cp
                        .cpItem(attributeNameIndex)).b);
                if (attributeName.equals(ATT_CONSTANT_VALUE)) {
                    attributeInfo = readConstantValueAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_CODE)) {
                    attributeInfo = readCodeAttribute(in, attributeNameIndex,
                            attributeLength);
                } else if (attributeName.equals(ATT_EXCEPTIONS)) {
                    attributeInfo = readExceptionsAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_INNER_CLASSES)) {
                    attributeInfo = readInnerClassesAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_ENCLOSING_METHOD)) {
                    attributeInfo = readEnclosingMethodAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_SYNTHETIC)) {
                    attributeInfo = readSyntheticAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_SIGNATURE)) {
                    attributeInfo = readSignatureAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_SOURCE_FILE)) {
                    attributeInfo = readSourceFileAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_SOURCE_DEBUG_EXTENSION)) {
                    attributeInfo = readSourceDebugExtensionAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_LINE_NUMBER_TABLE)) {
                    attributeInfo = readLineNumberTableAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_LOCAL_VARIABLE_TABLE)) {
                    attributeInfo = readLocalVariableTableAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_LOCAL_VARIABLE_TYPE_TABLE)) {
                    attributeInfo = readLocalVariableTypeTableAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_DEPRECATED)) {
                    attributeInfo = readDeprecatedAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName
                        .equals(ATT_RUNTIME_VISIBLE_ANNOTATIONS)) {
                    attributeInfo = readRuntimeVisibleAnnotationsAttribute(in,
                            attributeNameIndex, attributeLength);
                } else if (attributeName
                        .equals(ATT_RUNTIME_INVISIBLE_ANNOTATIONS)) {
                    attributeInfo = readRuntimeInvisibleAnnotationsAttribute(
                            in, attributeNameIndex, attributeLength);
                } else if (attributeName
                        .equals(ATT_RUNTIME_VISIBLE_PARAMETER_ANNOTATIONS)) {
                    attributeInfo =
readRuntimeVisibleParameterAnnotationsAttribute(
                            in, attributeNameIndex, attributeLength);
                } else if (attributeName
                        .equals(ATT_RUNTIME_INVISIBLE_PARAMETER_ANNOTATIONS)) {
                    attributeInfo =
readRuntimeInvisibleParameterAnnotationsAttribute(
                            in, attributeNameIndex, attributeLength);
                } else if (attributeName.equals(ATT_ANNOTATION_DEFAULT)) {
                    attributeInfo = readAnnotationDefaultAttribute(in,
                            attributeNameIndex, attributeLength);
                } else {
                    reportWarning("Unknown attribute '%s'", attributeName,
                            className);
                    for (long j = 0; j < attributeLength; j++) {
                        in.readUnsignedByte();
                    }
                }
```

```
                if (attributeInfo != null) {
                    attributes.add(attributeInfo);
                }
            }
        } catch (IOException e) {
            reportError("Error reading attributes from file %s", className);
        }
        return attributes;
    }

    /**
     * Read a ConstantValue attribute from the specified input stream, and
     * return it.
     *
     * @param in
     *            input stream.
     * @param attributeNameIndex
     *            constant pool index of the attribute name.
     * @param attributeLength
     *            length of attribute.
     * @return a ConstantValue attribute.
     */

    private CLConstantValueAttribute readConstantValueAttribute(
            CLInputStream in, int attributeNameIndex, long attributeLength) {
        CLConstantValueAttribute attribute = null;
        try {
            attribute = new CLConstantValueAttribute(attributeNameIndex,
                    attributeLength, in.readUnsignedShort());
        } catch (IOException e) {
            reportError("Error reading ConstantValue attribute from file %s",
                    className);
        }
        return attribute;
    }

    /**
     * Read a Code attribute from the specified input stream, and return it.
     *
     * @param in
     *            input stream.
     * @param attributeNameIndex
     *            constant pool index of the attribute name.
     * @param attributeLength
     *            length of attribute.
     * @return a Code attribute.
     */

    private CLCodeAttribute readCodeAttribute(CLInputStream in,
            int attributeNameIndex, long attributeLength) {
        CLCodeAttribute attribute = null;
        try {
            int maxStack = in.readUnsignedShort();
            int maxLocals = in.readUnsignedShort();
            ArrayList<Integer> code = new ArrayList<Integer>();
            long codeLength = in.readUnsignedInt();
            for (long l = 0; l < codeLength; l++) {
                code.add(in.readUnsignedByte());
            }
            int exceptionTableLength = in.readUnsignedShort();
            ArrayList<CLExceptionInfo> exceptionTable = new
ArrayList<CLExceptionInfo>();
            for (int l = 0; l < exceptionTableLength; l++) {
                int startPC = in.readUnsignedShort();
                int endPC = in.readUnsignedShort();
                int handlerPC = in.readUnsignedShort();
                int catchType = in.readUnsignedShort();
                exceptionTable.add(new CLExceptionInfo(startPC, endPC,
                        handlerPC, catchType));
```

```
            }
            int codeAttrAttributesCount = in.readUnsignedShort();
            ArrayList<CLAttributeInfo> codeAttrAttributes = readAttributes(in,
                    codeAttrAttributesCount);
            attribute = new CLCodeAttribute(attributeNameIndex,
                    attributeLength, maxStack, maxLocals, codeLength, code,
                    exceptionTableLength, exceptionTable,
                    codeAttrAttributesCount, codeAttrAttributes);
        } catch (IOException e) {
            reportError("Error reading Code_attribute from file %s", className);
        }
        return attribute;
    }

    /**
     * Read an Exceptions attribute from the specified input stream, and return
     * it.
     *
     * @param in
     *            input stream.
     * @param attributeNameIndex
     *            constant pool index of the attribute name.
     * @param attributeLength
     *            length of attribute.
     * @return an Exceptions attribute.
     */

    private CLExceptionsAttribute readExceptionsAttribute(CLInputStream in,
            int attributeNameIndex, long attributeLength) {
        CLExceptionsAttribute attribute = null;
        try {
            int numberOfExceptions = in.readUnsignedShort();
            ArrayList<Integer> exceptionIndexTable = new ArrayList<Integer>();
            for (int l = 0; l < numberOfExceptions; l++) {
                exceptionIndexTable.add(in.readUnsignedShort());
            }
            attribute = new CLExceptionsAttribute(attributeNameIndex,
                    attributeLength, numberOfExceptions, exceptionIndexTable);
        } catch (IOException e) {
            reportError("Error reading Exceptions_attribute from file %s",
                    className);
        }
        return attribute;
    }

    /**
     * Read an InnerClasses attribute from the specified input stream, and
     * return it.
     *
     * @param in
     *            input stream.
     * @param attributeNameIndex
     *            constant pool index of the attribute name.
     * @param attributeLength
     *            length of attribute.
     * @return an InnerClasses attribute.
     */

    private CLInnerClassesAttribute readInnerClassesAttribute(CLInputStream in,
            int attributeNameIndex, long attributeLength) {
        CLInnerClassesAttribute attribute = null;
        try {
            int numberOfClasses = in.readUnsignedShort();
            ArrayList<CLInnerClassInfo> classes = new
ArrayList<CLInnerClassInfo>();
            for (int m = 0; m < numberOfClasses; m++) {
                classes.add(new CLInnerClassInfo(in.readUnsignedShort(), in
                        .readUnsignedShort(), in.readUnsignedShort(), in
                        .readUnsignedShort()));
```

```
407             }
408             attribute = new CLInnerClassesAttribute(attributeNameIndex,
409                     attributeLength, numberOfClasses, classes);
410
411         } catch (IOException e) {
412             reportError("Error reading InnerClasses_attribute from file %s",
413                     className);
414         }
415         return attribute;
416     }
417
418     /**
419      * Read an EnclosingMethod attribute from the specified input stream, and
420      * return it.
421      *
422      * @param in
423      *            input stream.
424      * @param attributeNameIndex
425      *            constant pool index of the attribute name.
426      * @param attributeLength
427      *            length of attribute.
428      * @return an EnclosingMethod attribute.
429      */
430
431     private CLEnclosingMethodAttribute readEnclosingMethodAttribute(
432             CLInputStream in, int attributeNameIndex, long attributeLength) {
433         CLEnclosingMethodAttribute attribute = null;
434         try {
435             attribute = new CLEnclosingMethodAttribute(attributeNameIndex,
436                     attributeLength, in.readUnsignedShort(), in
437                             .readUnsignedShort());
438         } catch (IOException e) {
439             reportError("Error reading EnclosingMethod_attribute from file %s",
440                     className);
441         }
442         return attribute;
443     }
444
445     /**
446      * Read a Synthetic attribute from the specified input stream, and return
447      * it.
448      *
449      * @param in
450      *            input stream.
451      * @param attributeNameIndex
452      *            constant pool index of the attribute name.
453      * @param attributeLength
454      *            length of attribute.
455      * @return a Synthetic attribute.
456      */
457
458     private CLSyntheticAttribute readSyntheticAttribute(CLInputStream in,
459             int attributeNameIndex, long attributeLength) {
460         return new CLSyntheticAttribute(attributeNameIndex, attributeLength);
461     }
462
463     /**
464      * Read a Signature attribute from the specified input stream, and return
465      * it.
466      *
467      * @param in
468      *            input stream.
469      * @param attributeNameIndex
470      *            constant pool index of the attribute name.
471      * @param attributeLength
472      *            length of attribute.
473      * @return a Signature attribute.
474      */
475
```

```
476     private CLSignatureAttribute readSignatureAttribute(CLInputStream in,
477             int attributeNameIndex, long attributeLength) {
478         CLSignatureAttribute attribute = null;
479         try {
480             attribute = new CLSignatureAttribute(attributeNameIndex,
481                     attributeLength, in.readUnsignedShort());
482         } catch (IOException e) {
483             reportError("Error reading Signature_attribute from file %s",
484                     className);
485         }
486         return attribute;
487     }
488
489     /**
490      * Read a SourceFile attribute from the specified input stream, and return
491      * it.
492      *
493      * @param in
494      *            input stream.
495      * @param attributeNameIndex
496      *            constant pool index of the attribute name.
497      * @param attributeLength
498      *            length of attribute.
499      * @return a SourceFile attribute.
500      */
501
502     private CLSourceFileAttribute readSourceFileAttribute(CLInputStream in,
503             int attributeNameIndex, long attributeLength) {
504         CLSourceFileAttribute attribute = null;
505         try {
506             attribute = new CLSourceFileAttribute(attributeNameIndex,
507                     attributeLength, in.readUnsignedShort());
508         } catch (IOException e) {
509             reportError("Error reading SourceFile attribute from file %s",
510                     className);
511         }
512         return attribute;
513     }
514
515     /**
516      * Read a SourceDebugExtension attribute from the specified input stream,
517      * and return it.
518      *
519      * @param in
520      *            input stream.
521      * @param attributeNameIndex
522      *            constant pool index of the attribute name.
523      * @param attributeLength
524      *            length of attribute.
525      * @return a SourceDebugExtension attribute.
526      */
527
528     private CLSourceDebugExtensionAttribute readSourceDebugExtensionAttribute(
529             CLInputStream in, int attributeNameIndex, long attributeLength) {
530         CLSourceDebugExtensionAttribute attribute = null;
531         try {
532             byte[] b = new byte[(int) attributeLength];
533
534             in.read(b);
535             attribute = new CLSourceDebugExtensionAttribute(attributeNameIndex,
536                     attributeLength, b);
537         } catch (IOException e) {
538             reportError("Error reading SourceDebugExtension_attribute "
539                     + "from file %s", className);
540         }
541         return attribute;
542     }
543
544     /**
```

```
545      * Read a LineNumberTable attribute from the specified input stream, and
546      * return it.
547      *
548      * @param in
549      *            input stream.
550      * @param attributeNameIndex
551      *            constant pool index of the attribute name.
552      * @param attributeLength
553      *            length of attribute.
554      * @return a LineNumberTable attribute.
555      */
556
557     private CLLineNumberTableAttribute readLineNumberTableAttribute(
558             CLInputStream in, int attributeNameIndex, long attributeLength) {
559         CLLineNumberTableAttribute attribute = null;
560         try {
561             int lineNumberTableLength = in.readUnsignedShort();
562             ArrayList<CLLineNumberInfo> lineNumberTable = new
ArrayList<CLLineNumberInfo>();
563             for (int m = 0; m < lineNumberTableLength; m++) {
564                 lineNumberTable.add(new CLLineNumberInfo(
565                         in.readUnsignedShort(), in.readUnsignedShort()));
566             }
567             attribute = new CLLineNumberTableAttribute(attributeNameIndex,
568                     attributeLength, lineNumberTableLength, lineNumberTable);
569         } catch (IOException e) {
570             reportError("Error reading LineNumberTable_attribute from file %s",
571                     className);
572         }
573         return attribute;
574     }
575
576     /**
577      * Read a LocalVariableTable attribute from the specified input stream, and
578      * return it.
579      *
580      * @param in
581      *            input stream.
582      * @param attributeNameIndex
583      *            constant pool index of the attribute name.
584      * @param attributeLength
585      *            length of attribute.
586      * @return a LocalVariableTable attribute.
587      */
588
589     private CLLocalVariableTableAttribute readLocalVariableTableAttribute(
590             CLInputStream in, int attributeNameIndex, long attributeLength) {
591         CLLocalVariableTableAttribute attribute = null;
592         try {
593             int localVariableTableLength = in.readUnsignedShort();
594             ArrayList<CLLocalVariableInfo> localVariableTable = new
ArrayList<CLLocalVariableInfo>();
595             for (int m = 0; m < localVariableTableLength; m++) {
596                 localVariableTable.add(new CLLocalVariableInfo(in
597                         .readUnsignedShort(), in.readUnsignedShort(), in
598                         .readUnsignedShort(), in.readUnsignedShort(), in
599                         .readUnsignedShort()));
600             }
601             attribute = new CLLocalVariableTableAttribute(attributeNameIndex,
602                     attributeLength, localVariableTableLength,
603                     localVariableTable);
604         } catch (IOException e) {
605             reportError("Error reading LocalVariableTable_attribute "
606                     + "from file %s", className);
607         }
608         return attribute;
609     }
610
611     /**
```

```
612      * Read a LocalVariableTypeTable attribute from the specified input stream,
613      * and return it.
614      *
615      * @param in
616      *            input stream.
617      * @param attributeNameIndex
618      *            constant pool index of the attribute name.
619      * @param attributeLength
620      *            length of attribute.
621      * @return a LocalVariableTypeTable attribute.
622      */
623
624     private CLLocalVariableTypeTableAttribute
readLocalVariableTypeTableAttribute(
625             CLInputStream in, int attributeNameIndex, long attributeLength) {
626         CLLocalVariableTypeTableAttribute attribute = null;
627         try {
628             int localVariableTypeTableLength = in.readUnsignedShort();
629             ArrayList<CLLocalVariableTypeInfo> localVariableTypeTable = new
ArrayList<CLLocalVariableTypeInfo>();
630             for (int m = 0; m < localVariableTypeTableLength; m++) {
631                 localVariableTypeTable.add(new CLLocalVariableTypeInfo(in
632                         .readUnsignedShort(), in.readUnsignedShort(), in
633                         .readUnsignedShort(), in.readUnsignedShort(), in
634                         .readUnsignedShort()));
635             }
636             attribute = new CLLocalVariableTypeTableAttribute(
637                     attributeNameIndex, attributeLength,
638                     localVariableTypeTableLength, localVariableTypeTable);
639         } catch (IOException e) {
640             reportError("Error reading LocalVariableTypeTable_attribute"
641                     + "file %s", className);
642         }
643         return attribute;
644     }
645
646     /**
647      * Read a Deprecated attribute from the specified input stream, and return
648      * it.
649      *
650      * @param in
651      *            input stream.
652      * @param attributeNameIndex
653      *            constant pool index of the attribute name.
654      * @param attributeLength
655      *            length of attribute.
656      * @return a Deprecated attribute.
657      */
658
659     private CLDeprecatedAttribute readDeprecatedAttribute(CLInputStream in,
660             int attributeNameIndex, long attributeLength) {
661         return new CLDeprecatedAttribute(attributeNameIndex, attributeLength);
662     }
663
664     /**
665      * Read a RuntimeVisibleAnnotations attribute from the specified input
666      * stream, and return it.
667      *
668      * @param in
669      *            input stream.
670      * @param attributeNameIndex
671      *            constant pool index of the attribute name.
672      * @param attributeLength
673      *            length of attribute.
674      * @return a RuntimeVisibleAnnotations attribute.
675      */
676
677     private CLRuntimeVisibleAnnotationsAttribute
readRuntimeVisibleAnnotationsAttribute(
```

```
678             CLInputStream in, int attributeNameIndex, long attributeLength) {
679         CLRuntimeVisibleAnnotationsAttribute attribute = null;
680         try {
681             int numAnnotations = in.readUnsignedShort();
682             ArrayList<CLAnnotation> annotations = new ArrayList<CLAnnotation>();
683             for (int i = 0; i < numAnnotations; i++) {
684                 CLAnnotation annotation = readAnnotation(in);
685                 annotations.add(annotation);
686             }
687             attribute = new CLRuntimeVisibleAnnotationsAttribute(
688                     attributeNameIndex, attributeLength, numAnnotations,
689                     annotations);
690         } catch (IOException e) {
691             reportError("Error reading RuntimeVisibleAnnotations_attribute"
692                     + "from file %s", className);
693         }
694         return attribute;
695     }
696
697     /**
698      * Read a RuntimeInvisibleAnnotations attribute from the specified input
699      * stream, and return it.
700      *
701      * @param in
702      *             input stream.
703      * @param attributeNameIndex
704      *             constant pool index of the attribute name.
705      * @param attributeLength
706      *             length of attribute.
707      * @return a RuntimeInvisibleAnnotations attribute.
708      */
709
710     private CLRuntimeInvisibleAnnotationsAttribute
readRuntimeInvisibleAnnotationsAttribute(
711             CLInputStream in, int attributeNameIndex, long attributeLength) {
712         CLRuntimeInvisibleAnnotationsAttribute attribute = null;
713         try {
714             int numAnnotations = in.readUnsignedShort();
715             ArrayList<CLAnnotation> annotations = new ArrayList<CLAnnotation>();
716             for (int i = 0; i < numAnnotations; i++) {
717                 CLAnnotation annotation = readAnnotation(in);
718                 annotations.add(annotation);
719             }
720             attribute = new CLRuntimeInvisibleAnnotationsAttribute(
721                     attributeNameIndex, attributeLength, numAnnotations,
722                     annotations);
723         } catch (IOException e) {
724             reportError("Error reading RuntimeInvisibleAnnotations_attribute"
725                     + "from file %s", className);
726         }
727         return attribute;
728     }
729
730     /**
731      * Read a RuntimeVisibleParameterAnnotations attribute from the specified
732      * input stream, and return it.
733      *
734      * @param in
735      *             input stream.
736      * @param attributeNameIndex
737      *             constant pool index of the attribute name.
738      * @param attributeLength
739      *             length of attribute.
740      * @return a RuntimeVisibleParameterAnnotations attribute.
741      */
742
743     private CLRuntimeVisibleParameterAnnotationsAttribute
readRuntimeVisibleParameterAnnotationsAttribute(
744             CLInputStream in, int attributeNameIndex, long attributeLength) {
```

```
745         CLRuntimeVisibleParameterAnnotationsAttribute attribute = null;
746         try {
747             int numParameters = in.readUnsignedByte();
748             ArrayList<CLParameterAnnotationInfo> parameterAnnotations = new
ArrayList<CLParameterAnnotationInfo>();
749             for (int i = 0; i < numParameters; i++) {
750                 int numAnnotations = in.readUnsignedShort();
751                 ArrayList<CLAnnotation> annotations = new
ArrayList<CLAnnotation>();
752                 for (int j = 0; j < numAnnotations; j++) {
753                     CLAnnotation annotation = readAnnotation(in);
754                     annotations.add(annotation);
755                 }
756                 parameterAnnotations.add(new CLParameterAnnotationInfo(
757                         numAnnotations, annotations));
758             }
759             attribute = new CLRuntimeVisibleParameterAnnotationsAttribute(
760                     attributeNameIndex, attributeLength, (short) numParameters,
761                     parameterAnnotations);
762         } catch (IOException e) {
763             reportError("Error reading "
764                     + "RuntimeVisibleParameterAnnotations_attribute"
765                     + " from file %s", className);
766         }
767         return attribute;
768     }
769
770     /**
771      * Read a RuntimeInvisibleParameterAnnotations attribute from the specified
772      * input stream, and return it.
773      *
774      * @param in
775      *             input stream.
776      * @param attributeNameIndex
777      *             constant pool index of the attribute name.
778      * @param attributeLength
779      *             length of attribute.
780      * @return a RuntimeInvisibleParameterAnnotations attribute.
781      */
782
783     private CLRuntimeInvisibleParameterAnnotationsAttribute
readRuntimeInvisibleParameterAnnotationsAttribute(
784             CLInputStream in, int attributeNameIndex, long attributeLength) {
785         CLRuntimeInvisibleParameterAnnotationsAttribute attribute = null;
786         try {
787             int numParameters = in.readUnsignedByte();
788             ArrayList<CLParameterAnnotationInfo> parameterAnnotations = new
ArrayList<CLParameterAnnotationInfo>();
789             for (int i = 0; i < numParameters; i++) {
790                 int numAnnotations = in.readUnsignedShort();
791                 ArrayList<CLAnnotation> annotations = new
ArrayList<CLAnnotation>();
792                 for (int j = 0; j < numAnnotations; j++) {
793                     CLAnnotation annotation = readAnnotation(in);
794                     annotations.add(annotation);
795                 }
796                 parameterAnnotations.add(new CLParameterAnnotationInfo(
797                         numAnnotations, annotations));
798             }
799             attribute = new CLRuntimeInvisibleParameterAnnotationsAttribute(
800                     attributeNameIndex, attributeLength, (short) numParameters,
801                     parameterAnnotations);
802         } catch (IOException e) {
803             reportError("Error reading "
804                     + "RuntimeInvisibleParameterAnnotations_attribute"
805                     + " from file %s", className);
806         }
807         return attribute;
808     }
```

```
809
810     /**
811      * Read a AnnotationDefault attribute from t he specified input stream, and
812      * return it.
813      *
814      * @param in
815      *            input stream.
816      * @param attributeNameIndex
817      *            constant pool index of the attribute name.
818      * @param attributeLength
819      *            length of attribute.
820      * @return a AnnotationDefault attribute.
821      */
822
823     private CLAnnotationDefaultAttribute readAnnotationDefaultAttribute(
824             CLInputStream in, int attributeNameIndex, long attributeLength) {
825         return new CLAnnotationDefaultAttribute(attributeNameIndex,
826                 attributeLength, readElementValue(in));
827     }
828
829     /**
830      * Read an ElementValue from the specified input stream, and return it.
831      *
832      * @param in
833      *            input stream.
834      * @return an ElemenvValue.
835      */
836
837     private CLElementValue readElementValue(CLInputStream in) {
838         CLElementValue elementValue = null;
839         try {
840             int tag = in.readUnsignedByte();
841             switch (tag) {
842             case ELT_B:
843             case ELT_C:
844             case ELT_D:
845             case ELT_F:
846             case ELT_I:
847             case ELT_J:
848             case ELT_S:
849             case ELT_Z:
850             case ELT_s:
851                 elementValue = new CLElementValue((short) tag, in
852                         .readUnsignedShort());
853                 break;
854             case ELT_e:
855                 elementValue = new CLElementValue(in.readUnsignedShort(), in
856                         .readUnsignedShort());
857                 break;
858             case ELT_c:
859                 elementValue = new CLElementValue(in.readUnsignedShort());
860                 break;
861             case ELT_ANNOTATION:
862                 elementValue = new CLElementValue(readAnnotation(in));
863                 break;
864             case ELT_ARRAY:
865                 int numValues = in.readUnsignedShort();
866                 ArrayList<CLElementValue> values = new
ArrayList<CLElementValue>();
867                 for (int i = 0; i < numValues; i++) {
868                     values.add(readElementValue(in));
869                 }
870                 elementValue = new CLElementValue(numValues, values);
871             }
872         } catch (IOException e) {
873             reportError("Error reading AnnotationDefault_attribute "
874                     + "from file %s", className);
875         }
876         return elementValue;
```

```
877     }
878
879     /**
880      * Read an Annotation from the specified input stream, and return it.
881      *
882      * @param in
883      *            input stream.
884      * @return an Annotation.
885      */
886
887     private CLAnnotation readAnnotation(CLInputStream in) {
888         CLAnnotation annotation = null;
889         try {
890             int typeIndex = in.readUnsignedShort();
891             int numElementValuePairs = in.readUnsignedShort();
892             ArrayList<CLElementValuePair> elementValuePairs = new
ArrayList<CLElementValuePair>();
893             for (int i = 0; i < numElementValuePairs; i++) {
894                 int elementNameIndex = in.readUnsignedShort();
895                 CLElementValue value = readElementValue(in);
896                 elementValuePairs.add(new CLElementValuePair(elementNameIndex,
897                         value));
898             }
899             annotation = new CLAnnotation(typeIndex, numElementValuePairs,
900                     elementValuePairs);
901         } catch (IOException e) {
902             reportError("Error reading Annotation from file %s", className);
903         }
904         return annotation;
905     }
906
907     /**
908      * Construct a CLAbsorber object given the (fully-qualified) name of the
909      * class file to read.
910      *
911      * @param className
912      *            fully qualified name of the input class file.
913      */
914
915     public CLAbsorber(String className) {
916         try {
917             this.className = className;
918             CLPath classPath = new CLPath();
919             CLInputStream in = classPath.loadClass(className);
920             errorHasOccurred = false;
921             if (in == null) {
922                 reportError("Error loading %s", className);
923                 return;
924             }
925             classFile = new CLFile();
926
927             // Read magic number (0xCAFEBABE)
928             long magic = in.readUnsignedInt();
929             if (magic != MAGIC) {
930                 reportWarning("%s has an invalid magic number", className);
931                 return;
932             }
933             classFile.magic = magic;
934
935             // Read minor version, major version
936             classFile.minorVersion = in.readUnsignedShort();
937             classFile.majorVersion = in.readUnsignedShort();
938
939             // Read constant pool count, constant pool
940             classFile.constantPoolCount = in.readUnsignedShort();
941             classFile.constantPool = readConstantPool(in);
942             if (errorHasOccurred()) {
943                 return;
944             }
```

```
945
946             // Read access flags for the class
947             classFile.accessFlags = in.readUnsignedShort();
948
949             // Read this class' constant pool index
950             classFile.thisClass = in.readUnsignedShort();
951
952             // Read super class' constant pool index
953             classFile.superClass = in.readUnsignedShort();
954
955             // Read interfaces (implemented by the class being
956             // read) count and interfaces (constant pool
957             // indices)
958             int interfacesCount = in.readUnsignedShort();
959             ArrayList<Integer> interfaces = new ArrayList<Integer>();
960             classFile.interfacesCount = interfacesCount;
961             for (int i = 0; i < interfacesCount; i++) {
962                 interfaces.add(in.readUnsignedShort());
963             }
964             classFile.interfaces = interfaces;
965
966             // Read fields count and fields
967             classFile.fieldsCount = in.readUnsignedShort();
968             classFile.fields = readFields(in, classFile.fieldsCount);
969             if (errorHasOccurred()) {
970                 return;
971             }
972
973             // Read methods count and methods
974             classFile.methodsCount = in.readUnsignedShort();
975             classFile.methods = readMethods(in, classFile.methodsCount);
976             if (errorHasOccurred()) {
977                 return;
978             }
979
980             // Read class attributes
981             classFile.attributesCount = in.readUnsignedShort();
982             classFile.attributes = readAttributes(in, classFile.attributesCount);
983         } catch (EOFException e) {
984             reportError("Unexpected end of file %s", className);
985         } catch (IOException e) {
986             reportError("Error reading file %s", className);
987         }
988     }
989
990     /**
991      * Return the CLFile representation of the class that was read.
992      *
993      * @return the CLFile representation of the class.
994      */
995
996     public CLFile classFile() {
997         return classFile;
998     }
999
1000    /**
1001     * Return true if an error had occurred while reading the class; false
1002     * otherwise.
1003     *
1004     * @return true or false.
1005     */
1006
1007    public boolean errorHasOccurred() {
1008        return errorHasOccurred;
1009    }
1010
1011    /**
1012     * Driver for CLAbsorber. It accepts the (fully-qualified) name of a class
1013     * file as command-line argument and dumps its (ClassFile) structure --
```

```
     * CLFile in our representation -- to STDOUT in a format similar to that of
     * javap.
     */

    public static void main(String[] args) {
        String classFile = "";
        if (args.length == 1) {
            classFile = args[0];
        } else {
            String usage = "Usage: java jminusminus.CLAbsorber <class name>\n"
                    + "Where the class name must be fully qualified; "
                    + "eg, java/util/ArrayList";
            System.out.println(usage);
            System.exit(0);
        }
        CLAbsorber r = new CLAbsorber(classFile);
        if (!r.errorHasOccurred()) {
            CLFile c = r.classFile();
            c.writeToStdOut();
        }
    }

}

/**
 * Inherits from java.io.DataInputStream and provides an extra function for
 * reading unsigned int from the input stream, which is required for reading
 * Java class files.
 */

class CLInputStream extends DataInputStream {

    /**
     * Construct a CLInputStream object from the specified input stream.
     *
     * @param in
     *            input stream.
     */

    public CLInputStream(InputStream in) {
        super(in);
    }

    /**
     * Read four input bytes and return a long value in the range 0 through
     * 4294967295. Let a, b, c, d be the four bytes. The value returned is:
     *
     * <pre>
     *   ( b[ 0 ] &amp; 0xFF ) &lt;&lt; 24 ) |
     *   ( ( b[ 1 ] &amp; 0xFF ) &lt;&lt; 16 ) |
     *   ( ( b[ 2 ] &amp; 0xFF ) &lt;&lt; 8 ) |
     *   ( b[ 3 ] &amp; 0xFF )
     * </pre>
     *
     * @return the unsigned 32-bit value.
     * @exception EOFException
     *                if this stream reaches the end before reading all the
     *                bytes.
     * @exception IOException
     *                if an I/O error occurs.
     */

    public final long readUnsignedInt() throws IOException {
        byte[] b = new byte[4];
        long mask = 0xFF, l;
        in.read(b);
        l = ((b[0] & mask) << 24) | ((b[1] & mask) << 16)
                | ((b[2] & mask) << 8) | (b[3] & mask);
        return l;
```

```
1083    }
1084
1085}
1086
```