Code with better type

@PeterHilton

http://hilton.org.uk/

```
    2 HEADER-A; SIZE IS 68; CLASS IS AN.
    2 PAGE; SIZE IS 7; CLASS IS NUMERIC; SIGNED; ZERO SUPPRESS; LEAVING 1 PLACES.
    2 FILL-7; SIZE IS 4; CLASS IS AN.
 1 SECOND-HEADING; SIZE IS 120; CLASS IS AN.
WORKING-STORAGE SECTION.
 77 DEFFICIENCY; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 PRODUCTION-QUOTA; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 DIFFERENCE-FACTOR; SIZE IS 8; CLASS IS NUMERIC; SIGNED.
 77 PAGE-COUNT; SIZE IS 7; CLASS IS NUMERIC; SIGNED.
 77 LINE-COUNT; SIZE IS 4; CLASS IS NUMERIC; SIGNED.
 1 COMPILATION-DATE; SIZE IS 17; CLASS IS AN.
    2 SW; SIZE IS 1.
    2 DATE; SIZE IS 15.
    2 EM; SIZE IS 1.
CONSTANT SECTION.
 77 TWENTY-PERCENT; SIZE IS 2; CLASS IS NUMERIC; SIGNED; POINT LOCATION IS LEFT 1 PLACE; VALUE IS "2 ".
 77 TEN-PERCENT; SIZE IS 2; CLASS IS NUMERIC; SIGNED; POINT LOCATION IS LEFT 1 PLACE; VALUE IS "1 ".
 77 HEADER-B; SIZE IS 120; CLASS IS AN; VALUE IS "      PRODUCT NUMBER          PRODUCT      MEASURE      QUANTITY ON-
    HAND   EXPECTED TURNOVER    PRODUCTION QUOTA      ".
PROCEDURE DIVISION.
    00001. OPEN INPUT MASTER, TRANSACTION; OUTPUT NEW-MASTER, REPORT.
    00002. ACCEPT COMPILATION-DATE FROM PAPER-TAPE-READER.
    00003. MOVE ZEROES TO PAGE-COUNT, LINE-COUNT.
    00004. PERFORM 35 THRU 42.
    00005. READ MASTER; AT END GO TO 46.
```

```
          beq scroll

          lda speed
          eor #$01
          sta speed

; ---------------------------------------
; Ordinary scroll... (he, he - beat me
; it is probably the shortest 1x1 scroll
; routine on the world ;)

scroll    lda roll
          sec
          sbc speed
          bpl nzero
          and #$07
          sta roll

          ldy #$00
rewrite   lda scrline+1,y
          sta scrline,y
          iny
          cpy #$27
          bne rewrite
```

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i = * ( long * ) &y;                       // evil floating point
                                               // bit level hacking

    i = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y = * ( float * ) &i;
    y = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
// y = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                               // this can be removed
  return y;
}
```

Innovations in source code typography

Courier typeface, commissioned by IBM (1955)

Syntax highlighting - colour/bold (1985)

Courier New, introduced with Windows 3.1 (1992)

Consolas font, commissioned by Microsoft (2004)

Hasklig code font with ligatures, Ian Tuomi (2012)

Q_rsqrt

```
float
(float number)                                                    {
  long i;
  float x2, y;
  const float threehalfs = 1·5F;
  x2 = number * 0·5F;
  y  = number;
  i  = * ( long * ) &y;
  i  = 0x5f3759df - ( i ≫ 1 );
  y  = * ( float * ) &i;
  y  = y * ( threehalfs - ( x2 * y * y ) );
  // y  = y * ( threehalfs - ( x2 * y * y ) );
  return y;                                                       }
```

evil floating point bit level hacking

WTF?

1st iteration

2nd iteration, this can be removed

```java
/***
 * ASM: a very small and fast Java bytecode manipulation framework
 * Copyright (c) 2000-2011 INRIA, France Telecom
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the copyright holders nor the names of its
 *    contributors may be used to endorse or promote products derived from
 *    this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
 * THE POSSIBILITY OF SUCH DAMAGE.
 */
package org.springframework.asm;

import java.io.IOException;
import java.io.InputStream;


/**
 * A Java class parser to make a {@link ClassVisitor} visit an existing class.
 * This class parses a byte array conforming to the Java class file format and
 * calls the appropriate visit methods of a given class visitor for each field,
 * method and bytecode instruction encountered.
 *
 * @author Eric Bruneton
 * @author Eugene Kuleshov
 */
public class ClassReader {

    /**
     * True to enable signatures support.
     */
    static final boolean SIGNATURES = true;

    /**
     * True to enable annotations support.
     */
    static final boolean ANNOTATIONS = true;

    /**
     * True to enable stack map frames support.
     */
    static final boolean FRAMES = true;

    /**
     * True to enable bytecode writing support.
     */
    static final boolean WRITER = true;

    /**
     * True to enable JSR_W and GOTO_W support.
     */
    static final boolean RESIZE = true;

    /**
     * Flag to skip method code. If this class is set <code>CODE</code>
     * attribute won't be visited. This can be used, for example, to retrieve
     * annotations for methods and method parameters.
     */
    public static final int SKIP_CODE = 1;


    /**
     * Flag to skip the debug information in the class. If this flag is set the
     * debug information of the class is not visited, i.e. the
     * {@link MethodVisitor#visitLocalVariable visitLocalVariable} and
     * {@link MethodVisitor#visitLineNumber visitLineNumber} methods will not be
     * called.
     */
    public static final int SKIP_DEBUG = 2;

    /**
     * Flag to skip the stack map frames in the class. If this flag is set the
     * stack map frames of the class is not visited, i.e. the
     * {@link MethodVisitor#visitFrame visitFrame} method will not be called.
     * This flag is useful when the {@link ClassWriter#COMPUTE_FRAMES} option is
     * used: it avoids visiting frames that will be ignored and recomputed from
     * scratch in the class writer.
     */
    public static final int SKIP_FRAMES = 4;

    /**
     * Flag to expand the stack map frames. By default stack map frames are
     * visited in their original format (i.e. "expanded" for classes whose
     * version is less than V1_6, and "compressed" for the other classes). If
     * this flag is set, stack map frames are always visited in expanded format
     * (this option adds a decompression/recompression step in ClassReader and
     * ClassWriter which degrades performances quite a lot).
     */
    public static final int EXPAND_FRAMES = 8;

    /**
     * Flag to expand the ASM pseudo instructions into an equivalent sequence of
     * standard bytecode instructions. When resolving a forward jump it may
     * happen that the signed 2 bytes offset reserved for it is not sufficient
     * to store the bytecode offset. In this case the jump instruction is
     * replaced with a temporary ASM pseudo instruction using an unsigned 2
     * bytes offset (see Label#resolve). This internal flag is used to re-read
     * classes containing such instructions, in order to replace them with
     * standard instructions. In addition, when this flag is used, GOTO_W and
     * JSR_W are <i>not</i> converted into GOTO and JSR, to make sure that
     * infinite loops where a GOTO_W is replaced with a GOTO in ClassReader and
     * converted back to a GOTO_W in ClassWriter cannot occur.
     */
    static final int EXPAND_ASM_INSNS = 256;


    /**
     * The class to be parsed. <i>The content of this array must not be
     * modified. This field is intended for {@link Attribute} sub classes, and
     * is normally not needed by class generators or adapters.</i>
     */
    public final byte[] b;

    /**
     * The start index of each constant pool item in {@link #b b}, plus one. The
     * one byte offset skips the constant pool item tag that indicates its type.
     */
    private final int[] items;


    /**
     * The String objects corresponding to the CONSTANT_Utf8 items. This cache
     * avoids multiple parsing of a given CONSTANT_Utf8 constant pool item,
     * which GREATLY improves performances (by a factor 2 to 3). This caching
     * strategy could be extended to all constant pool items, but its benefit
     * would not be so great for these items (because they are much less
     * expensive to parse than CONSTANT_Utf8 items).
     */
    private final String[] strings;


    /**
     * Maximum length of the strings contained in the constant pool of the
     * class.
     */
    private final int maxStringLength;


    /**
     * Start index of the class header information (access, name...) in
     * {@link #b b}.
     */
    public final int header;


    // ------------------------------------------------------------------------
    // Constructors
    // ------------------------------------------------------------------------


    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param b
     *            the bytecode of the class to be read.
     */
    public ClassReader(final byte[] b) {
        this(b, 0, b.length);
    }


    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param b
     *            the bytecode of the class to be read.
     * @param off
     *            the start offset of the class data.
     * @param len
     *            the length of the class data.
     */
    public ClassReader(final byte[] b, final int off, final int len) {
        this.b = b;
        // checks the class version
                                    /* SPRING PATCH: REMOVED FOR FORWARD COMPATIBILITY WITH JDK 9
        if (readShort(off + 6) > Opcodes.V1_8) {
            throw new IllegalArgumentException();
        }
                                    */
        // parses the constant pool
        items = new int[readUnsignedShort(off + 8)];
        int n = items.length;
        strings = new String[n];
        int max = 0;
        int index = off + 10;
        for (int i = 1; i < n; ++i) {
            items[i] = index + 1;
            int size;
            switch (b[index]) {
            case ClassWriter.FIELD:
            case ClassWriter.METH:
            case ClassWriter.IMETH:
            case ClassWriter.INT:
            case ClassWriter.FLOAT:
            case ClassWriter.NAME_TYPE:
            case ClassWriter.INDY:
                size = 5;
                break;
            case ClassWriter.LONG:
            case ClassWriter.DOUBLE:
                size = 9;
                ++i;
                break;
            case ClassWriter.UTF8:
                size = 3 + readUnsignedShort(index + 1);
                if (size > max) {
                    max = size;
                }
                break;
            case ClassWriter.HANDLE:
                size = 4;
                break;
            // case ClassWriter.CLASS:
            // case ClassWriter.STR:
            // case ClassWriter.MTYPE
            default:
                size = 3;
                break;
            }
            index += size;
        }
        maxStringLength = max;
        // the class header information starts just after the constant pool
        header = index;
    }


    /**
     * Returns the class's access flags (see {@link Opcodes}). This value may
     * not reflect Deprecated and Synthetic flags when bytecode is before 1.5
     * and those flags are represented by attributes.
     *
     * @return the class access flags
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public int getAccess() {
        return readUnsignedShort(header);
    }


    /**
     * Returns the internal name of the class (see
     * {@link Type#getInternalName() getInternalName}).
     *
     * @return the internal class name
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public String getClassName() {
```

package org.springframework.asm;

public class

ClassReader

import java.io.IOException;
import java.io.InputStream;

```
{
    /**
     * True to enable signatures support.
     */
    static final boolean SIGNATURES = true;

    /**
     * True to enable annotations support.
     */
    static final boolean ANNOTATIONS = true;

    /**
     * True to enable stack map frames support.
     */
    static final boolean FRAMES = true;

    /**
     * True to enable bytecode writing support.
     */
    static final boolean WRITER = true;

    /**
     * True to enable JSR_W and GOTO_W support.
     */
    static final boolean RESIZE = true;

    /**
     * Flag to skip method code. If this class is set <code>CODE</code>
     * attribute won't be visited. This can be used, for example, to retrieve
     * annotations for methods and method parameters.
     */
    public static final int SKIP_CODE = 1;

    /**
     * Flag to skip the debug information in the class. If this flag is set the
     * debug information of the class is not visited, i.e. the
     * {@link MethodVisitor#visitLocalVariable visitLocalVariable} and
     * {@link MethodVisitor#visitLineNumber visitLineNumber} methods will not
be
     * called.
     */
    public static final int SKIP_DEBUG = 2;

    /**
     * Flag to skip the stack map frames in the class. If this flag is set the
     * stack map frames of the class is not visited, i.e. the
     * {@link MethodVisitor#visitFrame visitFrame} method will not be called.
     * This flag is useful when the {@link ClassWriter#COMPUTE_FRAMES}
option is
     * used: it avoids visiting frames that will be ignored and recomputed from
     * scratch in the class writer.
     */
    public static final int SKIP_FRAMES = 4;

    /**
     * Flag to expand the stack map frames. By default stack map frames are
     * visited in their original format (i.e. "expanded" for classes whose
     * version is less than V1_6, and "compressed" for the other classes). If
     * this flag is set, stack map frames are always visited in expanded format
     * (this option adds a decompression/recompression step in ClassReader and
     * ClassWriter which degrades performances quite a lot).
     */
    public static final int EXPAND_FRAMES = 8;

    /**
     * Flag to expand the ASM pseudo instructions into an equivalent sequence of
     * standard bytecode instructions. When resolving a forward jump it may
     * happen that the signed 2 bytes offset reserved for it is not sufficient
     * to store the bytecode offset. In this case the jump instruction is
     * replaced with a temporary ASM pseudo instruction using an unsigned 2
     * bytes offset (see Label#resolve). This internal flag is used to re-read
     * classes containing such instructions, in order to replace them with
     * standard instructions. In addition, when this flag is used, GOTO_W and
     * JSR_W are <i>not</i> converted into GOTO and JSR, to make sure that
     * infinite loops where a GOTO_W is replaced with a GOTO in ClassReader and
     * converted back to a GOTO_W in ClassWriter cannot occur.
     */
    static final int EXPAND_ASM_INSNS = 256;

    /**
     * The class to be parsed. <i>The content of this array must not be
     * modified. This field is intended for {@link Attribute} sub classes, and
     * is normally not needed by class generators or adapters.</i>
     */
    public final byte[] b;

    /**
     * The start index of each constant pool item in {@link #b b}, plus one. The
     * one byte offset skips the constant pool item tag that indicates its type.
     */
    private final int[] items;

    /**
     * The String objects corresponding to the CONSTANT_Utf8 items. This cache
     * avoids multiple parsing of a given CONSTANT_Utf8 constant pool item,
     * which GREATLY improves performances (by a factor 2 to 3). This caching
     * strategy could be extended to all constant pool items, but its benefit
     * would not be so great for these items (because they are much less
     * expensive to parse than CONSTANT_Utf8 items).
     */
    private final String[] strings;

    /**
     * Maximum length of the strings contained in the constant pool of the
     * class.
     */
    private final int maxStringLength;

    /**
     * Start index of the class header information (access, name...) in
     * {@link #b b}.
     */
    public final int header;

    // ------------------------------------------------------------------------
    // Constructors
    // ------------------------------------------------------------------------

    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param b
     *            the bytecode of the class to be read.
     */
    public ClassReader(final byte[] b) {
        this(b, 0, b.length);
    }

    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param b
     *            the bytecode of the class to be read.
     * @param off
     *            the start offset of the class data.
     * @param len
     *            the length of the class data.
     */
    public ClassReader(final byte[] b, final int off, final int len) {
        this.b = b;
        // checks the class version
                                    /* SPRING PATCH: REMOVED FOR FORWARD
COMPATIBILITY WITH JDK 9
        if (readShort(off + 6) > Opcodes.V1_8) {
            throw new IllegalArgumentException();
        }
                                    */
        // parses the constant pool
        items = new int[readUnsignedShort(off + 8)];
        int n = items.length;
        strings = new String[n];
        int max = 0;
        int index = off + 10;
        for (int i = 1; i < n; ++i) {
            items[i] = index + 1;
            int size;
            switch (b[index]) {
            case ClassWriter.FIELD:
            case ClassWriter.METH:
            case ClassWriter.IMETH:
            case ClassWriter.INT:
            case ClassWriter.FLOAT:
            case ClassWriter.NAME_TYPE:
            case ClassWriter.INDY:
                size = 5;
                break;
            case ClassWriter.LONG:
            case ClassWriter.DOUBLE:
                size = 9;
                ++i;
                break;
            case ClassWriter.UTF8:
                size = 3 + readUnsignedShort(index + 1);
                if (size > max) {
                    max = size;
                }
                break;
            case ClassWriter.HANDLE:
                size = 4;
                break;
            // case ClassWriter.CLASS:
            // case ClassWriter.STR:
            // case ClassWriter.MTYPE
            default:
                size = 3;
                break;
            }
            index += size;
        }
        maxStringLength = max;
        // the class header information starts just after the constant pool
        header = index;
    }

    /**
     * Returns the class's access flags (see {@link Opcodes}). This value may
     * not reflect Deprecated and Synthetic flags when bytecode is before 1.5
     * and those flags are represented by attributes.
     *
     * @return the class access flags
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public int getAccess() {
        return readUnsignedShort(header);
    }

    /**
     * Returns the internal name of the class (see
     * {@link Type#getInternalName() getInternalName}).
     *
     * @return the internal class name
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public String getClassName() {
        return readClass(header + 2, new char[maxStringLength]);
    }

    /**
     * Returns the internal of name of the super class (see
     * {@link Type#getInternalName() getInternalName}). For interfaces, the
     * super class is {@link Object}.
     *
     * @return the internal name of super class, or <tt>null</tt> for
     *         {@link Object} class.
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public String getSuperName() {
        return readClass(header + 4, new char[maxStringLength]);
    }

    /**
     * Returns the internal names of the class's interfaces (see
     * {@link Type#getInternalName() getInternalName}).
     *
     * @return the array of internal names for all implemented interfaces or
     *         <tt>null</tt>.
     *
     * @see ClassVisitor#visit(int, int, String, String, String, String[])
     */
    public String[] getInterfaces() {
        int index = header + 6;
        int n = readUnsignedShort(index);
        String[] interfaces = new String[n];
        if (n > 0) {
            char[] buf = new char[maxStringLength];
            for (int i = 0; i < n; ++i) {
                index += 2;
                interfaces[i] = readClass(index, buf);
            }
        }
        return interfaces;
    }

    /**
     * Copies the constant pool data into the given {@link ClassWriter}. Should
     * be called before the {@link #accept(ClassVisitor,int)} method.
     *
     * @param classWriter
     *            the {@link ClassWriter} to copy constant pool into.
     */
    void copyPool(final ClassWriter classWriter) {
        char[] buf = new char[maxStringLength];
        int ll = items.length;
        Item[] items2 = new Item[ll];
        for (int i = 1; i < ll; i++) {
            int index = items[i];
            int tag = b[index - 1];
            Item item = new Item(i);
            int nameType;
            switch (tag) {
            case ClassWriter.FIELD:
            case ClassWriter.METH:
            case ClassWriter.IMETH:
                nameType = items[readUnsignedShort(index + 2)];
                item.set(tag, readClass(index, buf), readUTF8(nameType, buf),
                        readUTF8(nameType + 2, buf));
                break;
            case ClassWriter.INT:
                item.set(readInt(index));
                break;
            case ClassWriter.FLOAT:
                item.set(Float.intBitsToFloat(readInt(index)));
                break;
            case ClassWriter.NAME_TYPE:
                item.set(tag, readUTF8(index, buf), readUTF8(index + 2, buf),
                        null);
                break;
            case ClassWriter.LONG:
                item.set(readLong(index));
                ++i;
                break;
            case ClassWriter.DOUBLE:
                item.set(Double.longBitsToDouble(readLong(index)));
                ++i;
                break;
            case ClassWriter.UTF8: {
                String s = strings[i];
                if (s == null) {
                    index = items[i];
                    s = strings[i] = readUTF(index + 2,
                            readUnsignedShort(index), buf);
                }
                item.set(tag, s, null, null);
                break;
            }
            case ClassWriter.HANDLE: {
                int fieldOrMethodRef = items[readUnsignedShort(index + 1)];
                nameType = items[readUnsignedShort(fieldOrMethodRef + 2)];
                item.set(ClassWriter.HANDLE_BASE + readByte(index),
                        readClass(fieldOrMethodRef, buf),
                        readUTF8(nameType, buf), readUTF8(nameType + 2, buf));
                break;
            }
            case ClassWriter.INDY:
                if (classWriter.bootstrapMethods == null) {
                    copyBootstrapMethods(classWriter, items2, buf);
                }
                nameType = items[readUnsignedShort(index + 2)];
                item.set(readUTF8(nameType, buf), readUTF8(nameType + 2, buf),
                        readUnsignedShort(index));
                break;
            // case ClassWriter.STR:
            // case ClassWriter.CLASS:
            // case ClassWriter.MTYPE
            default:
                item.set(tag, readUTF8(index, buf), null, null);
                break;
            }

            int index2 = item.hashCode % items2.length;
            item.next = items2[index2];
            items2[index2] = item;
        }

        int off = items[1] - 1;
        classWriter.pool.putByteArray(b, off, header - off);
        classWriter.items = items2;
        classWriter.threshold = (int) (0.75d * ll);
        classWriter.index = ll;
    }

    /**
     * Copies the bootstrap method data into the given {@link ClassWriter}.
     * Should be called before the {@link #accept(ClassVisitor,int)} method.
     *
     * @param classWriter
     *            the {@link ClassWriter} to copy bootstrap methods into.
     */
    private void copyBootstrapMethods(final ClassWriter classWriter,
            final Item[] items, final char[] c) {
        // finds the "BootstrapMethods" attribute
        int u = getAttributes();
        boolean found = false;
        for (int i = readUnsignedShort(u); i > 0; --i) {
            String attrName = readUTF8(u + 2, c);
            if ("BootstrapMethods".equals(attrName)) {
                found = true;
                break;
            }
            u += 6 + readInt(u + 4);
        }
        if (!found) {
            return;
        }
        // copies the bootstrap methods in the class writer
        int boostrapMethodCount = readUnsignedShort(u + 8);
        for (int j = 0, v = u + 10; j < boostrapMethodCount; j++) {
            int position = v - u - 10;
            int hashCode = readConst(readUnsignedShort(v), c).hashCode();
            for (int k = readUnsignedShort(v + 2); k > 0; --k) {
                hashCode ^= readConst(readUnsignedShort(v + 4), c).hashCode();
                v += 2;
            }
            v += 4;
            Item item = new Item(j);
            item.set(position, hashCode & 0x7FFFFFFF);
            int index = item.hashCode % items.length;
            item.next = items[index];
            items[index] = item;
        }
        int attrSize = readInt(u + 4);
        ByteVector bootstrapMethods = new ByteVector(attrSize + 62);
        bootstrapMethods.putByteArray(b, u + 10, attrSize - 2);
        classWriter.bootstrapMethodsCount = boostrapMethodCount;
        classWriter.bootstrapMethods = bootstrapMethods;
    }

    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param is
     *            an input stream from which to read the class.
     * @throws IOException
     *             if a problem occurs during reading.
     */
    public ClassReader(final InputStream is) throws IOException {
        this(readClass(is, false));
    }

    /**
     * Constructs a new {@link ClassReader} object.
     *
     * @param name
     *            the binary qualified name of the class to be read.
     * @throws IOException
     *             if an exception occurs during reading.
     */
    public ClassReader(final String name) throws IOException {
        this(readClass(
                ClassLoader.getSystemResourceAsStream(name.replace('.', '/')
                        + ".class"), true));
    }

    /**
     * Reads the bytecode of a class.
     *
     * @param is
     *            an input stream from which to read the class.
     * @param close
     *            true to close the input stream after reading.
     * @return the bytecode read from the given input stream.
     * @throws IOException
     *             if a problem occurs during reading.
     */
    private static byte[] readClass(final InputStream is, boolean close)
            throws IOException {
        if (is == null) {
            throw new IOException("Class not found");
        }
        try {
            byte[] b = new byte[is.available()];
            int len = 0;
            while (true) {
                int n = is.read(b, len, b.length - len);
                if (n == -1) {
                    if (len < b.length) {
                        byte[] c = new byte[len];
                        System.arraycopy(b, 0, c, 0, len);
                        b = c;
                    }
                    return b;
                }
                len += n;
                if (len == b.length) {
                    int last = is.read();
                    if (last < 0) {
                        return b;
                    }
                    byte[] c = new byte[b.length + 1000];
                    System.arraycopy(b, 0, c, 0, len);
                    c[len++] = (byte) last;
                    b = c;
                }
            }
        } finally {
            if (close) {
                is.close();
            }
        }
    }

    // ------------------------------------------------------------------------
    // Public methods
    // ------------------------------------------------------------------------

    /**
     * Makes the given visitor visit the Java class of this {@link ClassReader}
     * . This class is the one specified in the constructor (see
     * {@link #ClassReader(byte[]) ClassReader}).
     *
     * @param classVisitor
     *            the visitor that must visit this class.
     * @param flags
     *            option flags that can be used to modify the default behavior
     *            of this class. See {@link #SKIP_DEBUG}, {@link
#EXPAND_FRAMES}
     *            , {@link #SKIP_FRAMES}, {@link #SKIP_CODE}.
     */
    public void accept(final ClassVisitor classVisitor, final int flags) {
        accept(classVisitor, new Attribute[0], flags);
```

Every development team should have a designer who does the layout and typography for the code

@PeterHilton

http://hilton.org.uk/presentations/beautiful-code