```
**********************************************************
   14667 Thu Oct 27 04:15:54 2011
new/src/share/classes/java/lang/invoke/CallSite.java
**********************************************************
______unchanged_portion_omitted_
  82 </pre></blockquote>
  83  * @author John Rose, JSR 292 EG
  84  */
  85 abstract
  86 public class CallSite {
  87     static { MethodHandleImpl.initStatics(); }

  89     // Fields used only by the JVM.  Do not use or change.
  90     private MemberName vmmethod; // supplied by the JVM (ref. to calling method)
  91     private int        vmindex;  // supplied by the JVM (BCI within calling meth

  93     // The actual payload of this call site:
  94     /*package-private*/
  95     MethodHandle target;

  97     /**
  98      * Make a blank call site object with the given method type.
  99      * An initial target method is supplied which will throw
 100      * an {@link IllegalStateException} if called.
 101      * <p>
 102      * Before this {@code CallSite} object is returned from a bootstrap method,
 103      * it is usually provided with a more useful target method,
 104      * via a call to {@link CallSite#setTarget(MethodHandle) setTarget}.
 105      * @throws NullPointerException if the proposed type is null
 106      */
 107     /*package-private*/
 108     CallSite(MethodType type) {
 109         target = type.invokers().uninitializedCallSite();
 110     }

 112     /**
 113      * Make a call site object equipped with an initial target method handle.
 114      * @param target the method handle which will be the initial target of the c
 115      * @throws NullPointerException if the proposed target is null
 116      */
 117     /*package-private*/
 118     CallSite(MethodHandle target) {
 119         target.type();  // null check
 120         this.target = target;
 121     }

 123     /**
 124      * Make a call site object equipped with an initial target method handle.
 125      * @param targetType the desired type of the call site
 126      * @param createTargetHook a hook which will bind the call site to the targe
 127      * @throws WrongMethodTypeException if the hook cannot be invoked on the req
 128      *         or if the target returned by the hook is not of the given {@code
 129      * @throws NullPointerException if the hook returns a null value
 130      * @throws ClassCastException if the hook returns something other than a {@c
 131      * @throws Throwable anything else thrown by the the hook function
 132      */
 133     /*package-private*/
 134     CallSite(MethodType targetType, MethodHandle createTargetHook) throws Throwa
 135         this(targetType);
 136         ConstantCallSite selfCCS = (ConstantCallSite) this;
 137         MethodHandle boundTarget = (MethodHandle) createTargetHook.invokeWithArg
 138         checkTargetChange(this.target, boundTarget);
 139         this.target = boundTarget;
 140     }

 142     /**
```



```
 143      * Returns the type of this call site's target.
 144      * Although targets may change, any call site's type is permanent, and can n
 145      * The {@code setTarget} method enforces this invariant by refusing any new
 146      * not have the previous target's type.
 147      * @return the type of the current target, which is also the type of any fut
 148      */
 149     public MethodType type() {
 150         // warning:  do not call getTarget here, because CCS.getTarget can throw
 151         return target.type();
 152     }

 154     /** Called from JVM (or low-level Java code) after the BSM returns the newly
 155      *  The parameters are JVM-specific.
 156      */
 157     void initializeFromJVM(String name,
 158                            MethodType type,
 159                            MemberName callerMethod,
 160                            int        callerBCI) {
 161         if (this.vmmethod != null) {
 162             // FIXME
 163             throw new BootstrapMethodError("call site has already been linked to
 164         }
 165         if (!this.type().equals(type)) {
 166             throw wrongTargetType(target, type);
 167         }
 168         this.vmindex  = callerBCI;
 169         this.vmmethod = callerMethod;
 170     }

 172     /**
 173      * Returns the target method of the call site, according to the
 174      * behavior defined by this call site's specific class.
 175      * The immediate subclasses of {@code CallSite} document the
 176      * class-specific behaviors of this method.
 177      *
 178      * @return the current linkage state of the call site, its target method han
 179      * @see ConstantCallSite
 180      * @see VolatileCallSite
 181      * @see #setTarget
 182      * @see ConstantCallSite#getTarget
 183      * @see MutableCallSite#getTarget
 184      * @see VolatileCallSite#getTarget
 185      */
 186     public abstract MethodHandle getTarget();

 188     /**
 189      * Updates the target method of this call site, according to the
 190      * behavior defined by this call site's specific class.
 191      * The immediate subclasses of {@code CallSite} document the
 192      * class-specific behaviors of this method.
 193      * <p>
 194      * The type of the new target must be {@linkplain MethodType#equals equal to
 195      * the type of the old target.
 196      *
 197      * @param newTarget the new target
 198      * @throws NullPointerException if the proposed new target is null
 199      * @throws WrongMethodTypeException if the proposed new target
 200      *         has a method type that differs from the previous target
 201      * @see CallSite#getTarget
 202      * @see ConstantCallSite#setTarget
 203      * @see MutableCallSite#setTarget
 204      * @see VolatileCallSite#setTarget
 205      */
 206     public abstract void setTarget(MethodHandle newTarget);

 208     void checkTargetChange(MethodHandle oldTarget, MethodHandle newTarget) {
```

```
209         MethodType oldType = oldTarget.type();
210         MethodType newType = newTarget.type();  // null check!
211         if (!newType.equals(oldType))
212             throw wrongTargetType(newTarget, oldType);
213     }
214
215     private static WrongMethodTypeException wrongTargetType(MethodHandle target,
216         return new WrongMethodTypeException(String.valueOf(target)+" should be o
217     }
218
219     /**
220      * Produces a method handle equivalent to an invokedynamic instruction
221      * which has been linked to this call site.
222      * <p>
223      * This method is equivalent to the following code:
224      * <blockquote><pre>
225      * MethodHandle getTarget, invoker, result;
226      * getTarget = MethodHandles.publicLookup().bind(this, "getTarget", MethodTy
227      * invoker = MethodHandles.exactInvoker(this.type());
228      * result = MethodHandles.foldArguments(invoker, getTarget)
229      * </pre></blockquote>
230      *
231      * @return a method handle which always invokes this call site's current tar
232      */
233     public abstract MethodHandle dynamicInvoker();
234
235     /*non-public*/ MethodHandle makeDynamicInvoker() {
236         MethodHandle getTarget = MethodHandleImpl.bindReceiver(GET_TARGET, this)
237         MethodHandle invoker = MethodHandles.exactInvoker(this.type());
238         return MethodHandles.foldArguments(invoker, getTarget);
239     }
240
241     private static final MethodHandle GET_TARGET;
242     static {
243         try {
244             GET_TARGET = IMPL_LOOKUP.
245                 findVirtual(CallSite.class, "getTarget", MethodType.methodType(M
246         } catch (ReflectiveOperationException e) {
247             throw new InternalError(e);
248         }
249     }
250
251     /** This guy is rolled into the default target if a MethodType is supplied t
252     /*package-private*/
253     static Empty uninitializedCallSite() {
254         throw new IllegalStateException("uninitialized call site");
255     }
256
257     // unsafe stuff:
258     private static final Unsafe unsafe = Unsafe.getUnsafe();
259     private static final long TARGET_OFFSET;
260
261     static {
262         try {
263             TARGET_OFFSET = unsafe.objectFieldOffset(CallSite.class.getDeclaredF
264         } catch (Exception ex) { throw new Error(ex); }
265     }
266
267     /*package-private*/
268     void setTargetNormal(MethodHandle newTarget) {
269         MethodHandleNatives.setCallSiteTargetNormal(this, newTarget);
269         target = newTarget;
270     }
271     /*package-private*/
272     MethodHandle getTargetVolatile() {
273         return (MethodHandle) unsafe.getObjectVolatile(this, TARGET_OFFSET);
```



```
274     }
275     /*package-private*/
276     void setTargetVolatile(MethodHandle newTarget) {
277         MethodHandleNatives.setCallSiteTargetVolatile(this, newTarget);
277         unsafe.putObjectVolatile(this, TARGET_OFFSET, newTarget);
278     }
279
280     // this implements the upcall from the JVM, MethodHandleNatives.makeDynamicC
281     static CallSite makeSite(MethodHandle bootstrapMethod,
282                              // Callee information:
283                              String name, MethodType type,
284                              // Extra arguments for BSM, if any:
285                              Object info,
286                              // Caller information:
287                              MemberName callerMethod, int callerBCI) {
288         Class<?> callerClass = callerMethod.getDeclaringClass();
289         Object caller = IMPL_LOOKUP.in(callerClass);
290         CallSite site;
291         try {
292             Object binding;
293             info = maybeReBox(info);
294             if (info == null) {
295                 binding = bootstrapMethod.invoke(caller, name, type);
296             } else if (!info.getClass().isArray()) {
297                 binding = bootstrapMethod.invoke(caller, name, type, info);
298             } else {
299                 Object[] argv = (Object[]) info;
300                 maybeReBoxElements(argv);
301                 if (3 + argv.length > 255)
302                     throw new BootstrapMethodError("too many bootstrap method ar
303                 MethodType bsmType = bootstrapMethod.type();
304                 if (bsmType.parameterCount() == 4 && bsmType.parameterType(3) ==
305                     binding = bootstrapMethod.invoke(caller, name, type, argv);
306                 else
307                     binding = MethodHandles.spreadInvoker(bsmType, 3)
308                         .invoke(bootstrapMethod, caller, name, type, argv);
309             }
310             //System.out.println("BSM for "+name+type+" => "+binding);
311             if (binding instanceof CallSite) {
312                 site = (CallSite) binding;
313             }  else {
314                 throw new ClassCastException("bootstrap method failed to produce
315             }
316             if (!site.getTarget().type().equals(type))
317                 throw new WrongMethodTypeException("wrong type: "+site.getTarget
318         } catch (Throwable ex) {
319             BootstrapMethodError bex;
320             if (ex instanceof BootstrapMethodError)
321                 bex = (BootstrapMethodError) ex;
322             else
323                 bex = new BootstrapMethodError("call site initialization excepti
324             throw bex;
325         }
326         return site;
327     }
328
329     private static Object maybeReBox(Object x) {
330         if (x instanceof Integer) {
331             int xi = (int) x;
332             if (xi == (byte) xi)
333                 x = xi;  // must rebox; see JLS 5.1.7
334         }
335         return x;
336     }
337     private static void maybeReBoxElements(Object[] xa) {
338         for (int i = 0; i < xa.length; i++) {
```

```
339             xa[i] = maybeReBox(xa[i]);
340         }
341     }
342 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   17388 Thu Oct 27 04:15:55 2011
new/src/share/classes/java/lang/invoke/MethodHandleNatives.java
**********************************************************
```

```
/*
 * Copyright (c) 2008, 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.lang.invoke;

import java.lang.invoke.MethodHandles.Lookup;
import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Field;
import static java.lang.invoke.MethodHandleNatives.Constants.*;
import static java.lang.invoke.MethodHandles.Lookup.IMPL_LOOKUP;

/**
 * The JVM interface for the method handles package is all here.
 * This is an interface internal and private to an implemetantion of JSR 292.
 * <em>This class is not part of the JSR 292 standard.</em>
 * @author jrose
 */
class MethodHandleNatives {

    private MethodHandleNatives() { } // static only

    /// MethodName support

    static native void init(MemberName self, Object ref);
    static native void expand(MemberName self);
    static native void resolve(MemberName self, Class<?> caller);
    static native int getMembers(Class<?> defc, String matchName, String matchSi
            int matchFlags, Class<?> caller, int skip, MemberName[] results);

    /// MethodHandle support

    /** Initialize the method handle to adapt the call. */
    static native void init(AdapterMethodHandle self, MethodHandle target, int a
    /** Initialize the method handle to call the correct method, directly. */
    static native void init(BoundMethodHandle self, Object target, int argnum);
    /** Initialize the method handle to call as if by an invoke* instruction. */
    static native void init(DirectMethodHandle self, Object ref, boolean doDispa

    /** Initialize a method type, once per form. */
    static native void init(MethodType self);
```



```
    /** Tell the JVM about a class's bootstrap method. */
    static native void registerBootstrap(Class<?> caller, MethodHandle bootstrap

    /** Ask the JVM about a class's bootstrap method. */
    static native MethodHandle getBootstrap(Class<?> caller);

    /** Tell the JVM that we need to change the target of an invokedynamic. */
    static native void setCallSiteTarget(CallSite site, MethodHandle target);

    /** Fetch the vmtarget field.
     *  It will be sanitized as necessary to avoid exposing non-Java references.
     *  This routine is for debugging and reflection.
     */
    static native Object getTarget(MethodHandle self, int format);

    /** Fetch the name of the handled method, if available.
     *  This routine is for debugging and reflection.
     */
    static MemberName getMethodName(MethodHandle self) {
        return (MemberName) getTarget(self, ETF_METHOD_NAME);
    }

    /** Fetch the reflective version of the handled method, if available.
     */
    static AccessibleObject getTargetMethod(MethodHandle self) {
        return (AccessibleObject) getTarget(self, ETF_REFLECT_METHOD);
    }

    /** Fetch the target of this method handle.
     *  If it directly targets a method, return a MemberName for the method.
     *  If it is chained to another method handle, return that handle.
     */
    static Object getTargetInfo(MethodHandle self) {
        return getTarget(self, ETF_HANDLE_OR_METHOD_NAME);
    }

    static Object[] makeTarget(Class<?> defc, String name, String sig, int mods,
        return new Object[] { defc, name, sig, mods, refc };
    }

    /** Fetch MH-related JVM parameter.
     *  which=0 retrieves MethodHandlePushLimit
     *  which=1 retrieves stack slot push size (in address units)
     */
    static native int getConstant(int which);

    /** Java copy of MethodHandlePushLimit in range 2..255. */
    static final int JVM_PUSH_LIMIT;
    /** JVM stack motion (in words) after one slot is pushed, usually -1.
     */
    static final int JVM_STACK_MOVE_UNIT;

    /** Which conv-ops are implemented by the JVM? */
    static final int CONV_OP_IMPLEMENTED_MASK;
    /** Derived mode flag.  Only false on some old JVM implementations. */
    static final boolean HAVE_RICOCHET_FRAMES;

    static final int OP_ROT_ARGS_DOWN_LIMIT_BIAS;

    static final boolean COUNT_GWT;

    /// CallSite support

    /** Tell the JVM that we need to change the target of a CallSite. */
    static native void setCallSiteTargetNormal(CallSite site, MethodHandle targe
```

```
120     static native void setCallSiteTargetVolatile(CallSite site, MethodHandle tar

122 #endif /* ! codereview */
123     private static native void registerNatives();
124     static {
125         registerNatives();
126         int k;
127         JVM_PUSH_LIMIT              = getConstant(Constants.GC_JVM_PUSH_LIMIT);
128         JVM_STACK_MOVE_UNIT         = getConstant(Constants.GC_JVM_STACK_MOVE_UN
129         k                           = getConstant(Constants.GC_CONV_OP_IMPLEMENT
130         CONV_OP_IMPLEMENTED_MASK    = (k != 0) ? k : DEFAULT_CONV_OP_IMPLEMENTED
131         k                           = getConstant(Constants.GC_OP_ROT_ARGS_DOWN_
132         OP_ROT_ARGS_DOWN_LIMIT_BIAS = (k != 0) ? (byte)k : -1;
133         HAVE_RICOCHET_FRAMES        = (CONV_OP_IMPLEMENTED_MASK & (1<<OP_COLLECT
134         COUNT_GWT                   = getConstant(Constants.GC_COUNT_GWT) != 0;
135         //sun.reflect.Reflection.registerMethodsToFilter(MethodHandleImpl.class,
136     }

138     // All compile-time constants go here.
139     // There is an opportunity to check them against the JVM's idea of them.
140     static class Constants {
141         Constants() { } // static only
142         // MethodHandleImpl
143         static final int // for getConstant
144                 GC_JVM_PUSH_LIMIT = 0,
145                 GC_JVM_STACK_MOVE_UNIT = 1,
146                 GC_CONV_OP_IMPLEMENTED_MASK = 2,
147                 GC_OP_ROT_ARGS_DOWN_LIMIT_BIAS = 3,
148                 GC_COUNT_GWT = 4;
149         static final int
150                 ETF_HANDLE_OR_METHOD_NAME = 0, // all available data (immediate
151                 ETF_DIRECT_HANDLE         = 1, // ultimate method handle (will b
152                 ETF_METHOD_NAME           = 2, // ultimate method as MemberName
153                 ETF_REFLECT_METHOD        = 3; // ultimate method as java.lang.r

155         // MemberName
156         // The JVM uses values of -2 and above for vtable indexes.
157         // Field values are simple positive offsets.
158         // Ref: src/share/vm/oops/methodOop.hpp
159         // This value is negative enough to avoid such numbers,
160         // but not too negative.
161         static final int
162                 MN_IS_METHOD           = 0x00010000, // method (not constructor)
163                 MN_IS_CONSTRUCTOR      = 0x00020000, // constructor
164                 MN_IS_FIELD            = 0x00040000, // field
165                 MN_IS_TYPE             = 0x00080000, // nested type
166                 MN_SEARCH_SUPERCLASSES = 0x00100000, // for MHN.getMembers
167                 MN_SEARCH_INTERFACES   = 0x00200000, // for MHN.getMembers
168                 VM_INDEX_UNINITIALIZED = -99;

170         // BoundMethodHandle
171         /** Constants for decoding the vmargslot field, which contains 2 values.
172         static final int
173             ARG_SLOT_PUSH_SHIFT = 16,
174             ARG_SLOT_MASK = (1<<ARG_SLOT_PUSH_SHIFT)-1;

176         // AdapterMethodHandle
177         /** Conversions recognized by the JVM.
178          *  They must align with the constants in java.lang.invoke.AdapterMethod
179          *  in the JVM file hotspot/src/share/vm/classfile/javaClasses.hpp.
180          */
181         static final int
182             OP_RETYPE_ONLY   = 0x0, // no argument changes; straight retype
183             OP_RETYPE_RAW    = 0x1, // straight retype, trusted (void->int, Obje
184             OP_CHECK_CAST    = 0x2, // ref-to-ref conversion; requires a Class a
185             OP_PRIM_TO_PRIM  = 0x3, // converts from one primitive to another
```



```
186             OP_REF_TO_PRIM   = 0x4, // unboxes a wrapper to produce a primitive
187             OP_PRIM_TO_REF   = 0x5, // boxes a primitive into a wrapper
188             OP_SWAP_ARGS     = 0x6, // swap arguments (vminfo is 2nd arg)
189             OP_ROT_ARGS      = 0x7, // rotate arguments (vminfo is displaced arg
190             OP_DUP_ARGS      = 0x8, // duplicates one or more arguments (at TOS)
191             OP_DROP_ARGS     = 0x9, // remove one or more argument slots
192             OP_COLLECT_ARGS  = 0xA, // combine arguments using an auxiliary func
193             OP_SPREAD_ARGS   = 0xB, // expand in place a varargs array (of known
194             OP_FOLD_ARGS     = 0xC, // combine but do not remove arguments; prep
195             //OP_UNUSED_13   = 0xD, // unused code, perhaps for reified argument
196             CONV_OP_LIMIT    = 0xE; // limit of CONV_OP enumeration
197         /** Shift and mask values for decoding the AMH.conversion field.
198          *  These numbers are shared with the JVM for creating AMHs.
199          */
200         static final int
201             CONV_OP_MASK     = 0xF00, // this nybble contains the conversion op
202             CONV_TYPE_MASK   = 0x0F,  // fits T_ADDRESS and below
203             CONV_VMINFO_MASK = 0x0FF, // LSB is reserved for JVM use
204             CONV_VMINFO_SHIFT     =  0, // position of bits in CONV_VMINFO_MASK
205             CONV_OP_SHIFT         =  8, // position of bits in CONV_OP_MASK
206             CONV_DEST_TYPE_SHIFT  = 12, // byte 2 has the adapter BasicType (if
207             CONV_SRC_TYPE_SHIFT   = 16, // byte 2 has the source BasicType (if n
208             CONV_STACK_MOVE_SHIFT = 20, // high 12 bits give signed SP change
209             CONV_STACK_MOVE_MASK  = (1 << (32 - CONV_STACK_MOVE_SHIFT)) - 1;

211         /** Which conv-ops are implemented by the JVM? */
212         static final int DEFAULT_CONV_OP_IMPLEMENTED_MASK =
213                 // Value to use if the corresponding JVM query fails.
214                 ((1<<OP_RETYPE_ONLY)
215                 |(1<<OP_RETYPE_RAW)
216                 |(1<<OP_CHECK_CAST)
217                 |(1<<OP_PRIM_TO_PRIM)
218                 |(1<<OP_REF_TO_PRIM)
219                 |(1<<OP_SWAP_ARGS)
220                 |(1<<OP_ROT_ARGS)
221                 |(1<<OP_DUP_ARGS)
222                 |(1<<OP_DROP_ARGS)
223                 //|(1<<OP_SPREAD_ARGS)
224                 );

226         /**
227          * Basic types as encoded in the JVM.  These code values are not
228          * intended for use outside this class.  They are used as part of
229          * a private interface between the JVM and this class.
230          */
231         static final int
232             T_BOOLEAN  =  4,
233             T_CHAR     =  5,
234             T_FLOAT    =  6,
235             T_DOUBLE   =  7,
236             T_BYTE     =  8,
237             T_SHORT    =  9,
238             T_INT      = 10,
239             T_LONG     = 11,
240             T_OBJECT   = 12,
241             //T_ARRAY    = 13
242             T_VOID     = 14,
243             //T_ADDRESS  = 15
244             T_ILLEGAL  = 99;

246         /**
247          * Constant pool reference-kind codes, as used by CONSTANT_MethodHandle
248          */
249         static final int
250             REF_getField               = 1,
251             REF_getStatic              = 2,
```

```
252             REF_putField                = 3,
253             REF_putStatic               = 4,
254             REF_invokeVirtual           = 5,
255             REF_invokeStatic            = 6,
256             REF_invokeSpecial           = 7,
257             REF_newInvokeSpecial        = 8,
258             REF_invokeInterface         = 9;
259     }

261     private static native int getNamedCon(int which, Object[] name);
262     static boolean verifyConstants() {
263         Object[] box = { null };
264         for (int i = 0; ; i++) {
265             box[0] = null;
266             int vmval = getNamedCon(i, box);
267             if (box[0] == null)  break;
268             String name = (String) box[0];
269             try {
270                 Field con = Constants.class.getDeclaredField(name);
271                 int jval = con.getInt(null);
272                 if (jval == vmval)  continue;
273                 String err = (name+": JVM has "+vmval+" while Java has "+jval);
274                 if (name.equals("CONV_OP_LIMIT")) {
275                     System.err.println("warning: "+err);
276                     continue;
277                 }
278                 throw new InternalError(err);
279             } catch (Exception ex) {
280                 if (ex instanceof NoSuchFieldException) {
281                     String err = (name+": JVM has "+vmval+" which Java does not
282                     // ignore exotic ops the JVM cares about; we just wont issue
283                     if (name.startsWith("OP_") || name.startsWith("GC_")) {
284                         System.err.println("warning: "+err);
285                         continue;
286                     }
287                 }
288                 throw new InternalError(name+": access failed, got "+ex);
289             }
290         }
291         return true;
292     }
293     static {
294         assert(verifyConstants());
295     }

297     // Up-calls from the JVM.
298     // These must NOT be public.

300     /**
301      * The JVM is linking an invokedynamic instruction.  Create a reified call s
302      */
303     static CallSite makeDynamicCallSite(MethodHandle bootstrapMethod,
304                                         String name, MethodType type,
305                                         Object info,
306                                         MemberName callerMethod, int callerBCI)
307         return CallSite.makeSite(bootstrapMethod, name, type, info, callerMethod
308     }

310     /**
311      * Called by the JVM to check the length of a spread array.
312      */
313     static void checkSpreadArgument(Object av, int n) {
314         MethodHandleStatics.checkSpreadArgument(av, n);
315     }

317     /**
```



```
318      * The JVM wants a pointer to a MethodType.  Oblige it by finding or creatin
319      */
320     static MethodType findMethodHandleType(Class<?> rtype, Class<?>[] ptypes) {
321         return MethodType.makeImpl(rtype, ptypes, true);
322     }

324     /**
325      * The JVM wants to use a MethodType with inexact invoke.  Give the runtime
326      */
327     static void notifyGenericMethodType(MethodType type) {
328         type.form().notifyGenericMethodType();
329     }

331     /**
332      * The JVM wants to raise an exception.  Here's the path.
333      */
334     static void raiseException(int code, Object actual, Object required) {
335         String message = null;
336         switch (code) {
337         case 190: // arraylength
338             try {
339                 String reqLength = "";
340                 if (required instanceof AdapterMethodHandle) {
341                     int conv = ((AdapterMethodHandle)required).getConversion();
342                     int spChange = AdapterMethodHandle.extractStackMove(conv);
343                     reqLength = " of length "+(spChange+1);
344                 }
345                 int actualLength = actual == null ? 0 : java.lang.reflect.Array.
346                 message = "required array"+reqLength+", but encountered wrong le
347                 break;
348             } catch (IllegalArgumentException ex) {
349             }
350             required = Object[].class;  // should have been an array
351             code = 192; // checkcast
352             break;
353         case 191: // athrow
354             // JVM is asking us to wrap an exception which happened during resol
355             if (required == BootstrapMethodError.class) {
356                 throw new BootstrapMethodError((Throwable) actual);
357             }
358             break;
359         }
360         // disregard the identity of the actual object, if it is not a class:
361         if (message == null) {
362             if (!(actual instanceof Class) && !(actual instanceof MethodType))
363                 actual = actual.getClass();
364            if (actual != null)
365                message = "required "+required+" but encountered "+actual;
366            else
367                message = "required "+required;
368         }
369         switch (code) {
370         case 190: // arraylength
371             throw new ArrayIndexOutOfBoundsException(message);
372         case 50: //_aaload
373             throw new ClassCastException(message);
374         case 192: // checkcast
375             throw new ClassCastException(message);
376         default:
377             throw new InternalError("unexpected code "+code+": "+message);
378         }
379     }

381     /**
382      * The JVM is resolving a CONSTANT_MethodHandle CP entry.  And it wants our
383      * It will make an up-call to this method.  (Do not change the name or signa
```

```
384      */
385     static MethodHandle linkMethodHandleConstant(Class<?> callerClass, int refKi
386                                                  Class<?> defc, String name, Obj
387         try {
388             Lookup lookup = IMPL_LOOKUP.in(callerClass);
389             return lookup.linkMethodHandleConstant(refKind, defc, name, type);
390         } catch (ReflectiveOperationException ex) {
391             Error err = new IncompatibleClassChangeError();
392             err.initCause(ex);
393             throw err;
394         }
395     }
396 }
```

```
*****************************************************
    5607 Thu Oct 27 04:15:57 2011
new/test/java/lang/invoke/CallSiteTest.java
*****************************************************
```

```
/*
 * Copyright (c) 2011, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 *
 */

/**
 * @test
 * @summary smoke tests for CallSite
 *
 * @build indify.Indify
 * @compile CallSiteTest.java
 * @run main/othervm
 *      indify.Indify
 *      --expand-properties --classpath ${test.classes}
 *      --java test.java.lang.invoke.CallSiteTest
 */

package test.java.lang.invoke;

import static org.junit.Assert.*;

import java.io.*;

import java.lang.invoke.*;
import static java.lang.invoke.MethodHandles.*;
import static java.lang.invoke.MethodType.*;

public class CallSiteTest {
    private final static Class CLASS = CallSiteTest.class;

    private static CallSite mcs;
    private static CallSite vcs;
    private static MethodHandle mh_foo;
    private static MethodHandle mh_bar;

    static {
        try {
            mh_foo = lookup().findStatic(CLASS, "foo", methodType(int.class, int
            mh_bar = lookup().findStatic(CLASS, "bar", methodType(int.class, int
            mcs = new MutableCallSite(mh_foo);
            vcs = new VolatileCallSite(mh_foo);
        } catch (Exception e) {
            e.printStackTrace();
```



```
        }
    }

    public static void main(String... av) throws Throwable {
        testMutableCallSite();
        testVolatileCallSite();
    }

    private final static int N = Integer.MAX_VALUE / 100;
    private final static int RESULT1 = 762786192;
    private final static int RESULT2 = -21474836;

    private static void testMutableCallSite() throws Throwable {
        // warm-up
        for (int i = 0; i < 20000; i++) {
            mcs.setTarget(mh_foo);
        }
        // run
        for (int n = 0; n < 2; n++) {
            mcs.setTarget(mh_foo);
            for (int i = 0; i < 5; i++) {
                assertEquals(RESULT1, runMutableCallSite());
            }
            mcs.setTarget(mh_bar);
            for (int i = 0; i < 5; i++) {
                assertEquals(RESULT2, runMutableCallSite());
            }
        }
    }
    private static void testVolatileCallSite() throws Throwable {
        // warm-up
        for (int i = 0; i < 20000; i++) {
            vcs.setTarget(mh_foo);
        }
        // run
        for (int n = 0; n < 2; n++) {
            vcs.setTarget(mh_foo);
            for (int i = 0; i < 5; i++) {
                assertEquals(RESULT1, runVolatileCallSite());
            }
            vcs.setTarget(mh_bar);
            for (int i = 0; i < 5; i++) {
                assertEquals(RESULT2, runVolatileCallSite());
            }
        }
    }

    private static int runMutableCallSite() throws Throwable {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += (int) INDY_mcs().invokeExact(i, i+1);
        }
        return sum;
    }
    private static int runVolatileCallSite() throws Throwable {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += (int) INDY_vcs().invokeExact(i, i+1);
        }
        return sum;
    }

    static int foo(int a, int b) { return a + b; }
    static int bar(int a, int b) { return a - b; }

    private static MethodType MT_bsm() {
```

```
129         shouldNotCallThis();
130         return methodType(CallSite.class, Lookup.class, String.class, MethodType
131     }

133     private static CallSite bsm_mcs(Lookup caller, String name, MethodType type)
134         return mcs;
135     }
136     private static MethodHandle MH_bsm_mcs() throws ReflectiveOperationException
137         shouldNotCallThis();
138         return lookup().findStatic(lookup().lookupClass(), "bsm_mcs", MT_bsm());
139     }
140     private static MethodHandle INDY_mcs() throws Throwable {
141         shouldNotCallThis();
142         return ((CallSite) MH_bsm_mcs().invoke(lookup(), "foo", methodType(int.c
143     }

145     private static CallSite bsm_vcs(Lookup caller, String name, MethodType type)
146         return vcs;
147     }
148     private static MethodHandle MH_bsm_vcs() throws ReflectiveOperationException
149         shouldNotCallThis();
150         return lookup().findStatic(lookup().lookupClass(), "bsm_vcs", MT_bsm());
151     }
152     private static MethodHandle INDY_vcs() throws Throwable {
153         shouldNotCallThis();
154         return ((CallSite) MH_bsm_vcs().invoke(lookup(), "foo", methodType(int.c
155     }

157     private static void shouldNotCallThis() {
158         // if this gets called, the transformation has not taken place
159         throw new AssertionError("this code should be statically transformed awa
160     }
161 }
162 #endif /* ! codereview */
```