001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017  
018package org.apache.commons.beanutils;  
019  
020  
021import java.lang.ref.Reference;  
022import java.lang.ref.WeakReference;  
023import java.lang.reflect.InvocationTargetException;  
024import java.lang.reflect.Method;  
025import java.lang.reflect.Modifier;  
026import java.util.Collections;  
027import java.util.Map;  
028import java.util.WeakHashMap;  
029  
030import org.apache.commons.logging.Log;  
031import org.apache.commons.logging.LogFactory;  
032  
033  
034/\*\*  
035 \* <p> Utility reflection methods focused on methods in general rather than properties in particular. </p>  
036 \*  
037 \* <h3>Known Limitations</h3>  
038 \* <h4>Accessing Public Methods In A Default Access Superclass</h4>  
039 \* <p>There is an issue when invoking public methods contained in a default access superclass.  
040 \* Reflection locates these methods fine and correctly assigns them as public.  
041 \* However, an <code>IllegalAccessException</code> is thrown if the method is invoked.</p>  
042 \*  
043 \* <p><code>MethodUtils</code> contains a workaround for this situation.  
044 \* It will attempt to call <code>setAccessible</code> on this method.  
045 \* If this call succeeds, then the method can be invoked as normal.  
046 \* This call will only succeed when the application has sufficient security privilages.  
047 \* If this call fails then a warning will be logged and the method may fail.</p>  
048 \*  
049 \* @version $Id$  
050 \*/  
051  
052public class MethodUtils {  
053  
054 // --------------------------------------------------------- Private Methods  
055  
056 /\*\*  
057 \* Only log warning about accessibility work around once.  
058 \* <p>  
059 \* Note that this is broken when this class is deployed via a shared  
060 \* classloader in a container, as the warning message will be emitted  
061 \* only once, not once per webapp. However making the warning appear  
062 \* once per webapp means having a map keyed by context classloader  
063 \* which introduces nasty memory-leak problems. As this warning is  
064 \* really optional we can ignore this problem; only one of the webapps  
065 \* will get the warning in its logs but that should be good enough.  
066 \*/  
067 private static boolean loggedAccessibleWarning = false;  
068  
069 /\*\*  
070 \* Indicates whether methods should be cached for improved performance.  
071 \* <p>  
072 \* Note that when this class is deployed via a shared classloader in  
073 \* a container, this will affect all webapps. However making this  
074 \* configurable per webapp would mean having a map keyed by context classloader  
075 \* which may introduce memory-leak problems.  
076 \*/  
077 private static boolean CACHE\_METHODS = true;  
078  
079 /\*\* An empty class array \*/  
080 private static final Class<?>[] EMPTY\_CLASS\_PARAMETERS = new Class[0];  
081 /\*\* An empty object array \*/  
082 private static final Object[] EMPTY\_OBJECT\_ARRAY = new Object[0];  
083  
084 /\*\*  
085 \* Stores a cache of MethodDescriptor -> Method in a WeakHashMap.  
086 \* <p>  
087 \* The keys into this map only ever exist as temporary variables within  
088 \* methods of this class, and are never exposed to users of this class.  
089 \* This means that the WeakHashMap is used only as a mechanism for  
090 \* limiting the size of the cache, ie a way to tell the garbage collector  
091 \* that the contents of the cache can be completely garbage-collected  
092 \* whenever it needs the memory. Whether this is a good approach to  
093 \* this problem is doubtful; something like the commons-collections  
094 \* LRUMap may be more appropriate (though of course selecting an  
095 \* appropriate size is an issue).  
096 \* <p>  
097 \* This static variable is safe even when this code is deployed via a  
098 \* shared classloader because it is keyed via a MethodDescriptor object  
099 \* which has a Class as one of its members and that member is used in  
100 \* the MethodDescriptor.equals method. So two components that load the same  
101 \* class via different classloaders will generate non-equal MethodDescriptor  
102 \* objects and hence end up with different entries in the map.  
103 \*/  
104 private static final Map<MethodDescriptor, Reference<Method>> cache = Collections  
105 .synchronizedMap(new WeakHashMap<MethodDescriptor, Reference<Method>>());  
106  
107 // --------------------------------------------------------- Public Methods  
108  
109 /\*\*  
110 \* Set whether methods should be cached for greater performance or not,  
111 \* default is <code>true</code>.  
112 \*  
113 \* @param cacheMethods <code>true</code> if methods should be  
114 \* cached for greater performance, otherwise <code>false</code>  
115 \* @since 1.8.0  
116 \*/  
117 public static synchronized void setCacheMethods(final boolean cacheMethods) {  
118 CACHE\_METHODS = cacheMethods;  
119 if (!CACHE\_METHODS) {  
120 clearCache();  
121 }  
122 }  
123  
124 /\*\*  
125 \* Clear the method cache.  
126 \* @return the number of cached methods cleared  
127 \* @since 1.8.0  
128 \*/  
129 public static synchronized int clearCache() {  
130 final int size = cache.size();  
131 cache.clear();  
132 return size;  
133 }  
134  
135 /\*\*  
136 \* <p>Invoke a named method whose parameter type matches the object type.</p>  
137 \*  
138 \* <p>The behaviour of this method is less deterministic  
139 \* than <code>invokeExactMethod()</code>.  
140 \* It loops through all methods with names that match  
141 \* and then executes the first it finds with compatible parameters.</p>  
142 \*  
143 \* <p>This method supports calls to methods taking primitive parameters  
144 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
145 \* would match a <code>boolean</code> primitive.</p>  
146 \*  
147 \* <p> This is a convenient wrapper for  
148 \* {@link #invokeMethod(Object object,String methodName,Object [] args)}.  
149 \* </p>  
150 \*  
151 \* @param object invoke method on this object  
152 \* @param methodName get method with this name  
153 \* @param arg use this argument. May be null (this will result in calling the  
154 \* parameterless method with name {@code methodName}).  
155 \* @return The value returned by the invoked method  
156 \*  
157 \* @throws NoSuchMethodException if there is no such accessible method  
158 \* @throws InvocationTargetException wraps an exception thrown by the  
159 \* method invoked  
160 \* @throws IllegalAccessException if the requested method is not accessible  
161 \* via reflection  
162 \*/  
163 public static Object invokeMethod(  
164 final Object object,  
165 final String methodName,  
166 final Object arg)  
167 throws  
168 NoSuchMethodException,  
169 IllegalAccessException,  
170 InvocationTargetException {  
171  
172 final Object[] args = toArray(arg);  
173 return invokeMethod(object, methodName, args);  
174 }  
175  
176  
177 /\*\*  
178 \* <p>Invoke a named method whose parameter type matches the object type.</p>  
179 \*  
180 \* <p>The behaviour of this method is less deterministic  
181 \* than {@link #invokeExactMethod(Object object,String methodName,Object [] args)}.  
182 \* It loops through all methods with names that match  
183 \* and then executes the first it finds with compatible parameters.</p>  
184 \*  
185 \* <p>This method supports calls to methods taking primitive parameters  
186 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
187 \* would match a <code>boolean</code> primitive.</p>  
188 \*  
189 \* <p> This is a convenient wrapper for  
190 \* {@link #invokeMethod(Object object,String methodName,Object [] args,Class[] parameterTypes)}.  
191 \* </p>  
192 \*  
193 \* @param object invoke method on this object  
194 \* @param methodName get method with this name  
195 \* @param args use these arguments - treat null as empty array (passing null will  
196 \* result in calling the parameterless method with name {@code methodName}).  
197 \* @return The value returned by the invoked method  
198 \*  
199 \* @throws NoSuchMethodException if there is no such accessible method  
200 \* @throws InvocationTargetException wraps an exception thrown by the  
201 \* method invoked  
202 \* @throws IllegalAccessException if the requested method is not accessible  
203 \* via reflection  
204 \*/  
205 public static Object invokeMethod(  
206 final Object object,  
207 final String methodName,  
208 Object[] args)  
209 throws  
210 NoSuchMethodException,  
211 IllegalAccessException,  
212 InvocationTargetException {  
213  
214 if (args == null) {  
215 args = EMPTY\_OBJECT\_ARRAY;  
216 }  
217 final int arguments = args.length;  
218 final Class<?>[] parameterTypes = new Class[arguments];  
219 for (int i = 0; i < arguments; i++) {  
220 parameterTypes[i] = args[i].getClass();  
221 }  
222 return invokeMethod(object, methodName, args, parameterTypes);  
223 }  
224  
225  
226 /\*\*  
227 \* <p>Invoke a named method whose parameter type matches the object type.</p>  
228 \*  
229 \* <p>The behaviour of this method is less deterministic  
230 \* than {@link  
231 \* #invokeExactMethod(Object object,String methodName,Object [] args,Class[] parameterTypes)}.  
232 \* It loops through all methods with names that match  
233 \* and then executes the first it finds with compatible parameters.</p>  
234 \*  
235 \* <p>This method supports calls to methods taking primitive parameters  
236 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
237 \* would match a <code>boolean</code> primitive.</p>  
238 \*  
239 \*  
240 \* @param object invoke method on this object  
241 \* @param methodName get method with this name  
242 \* @param args use these arguments - treat null as empty array (passing null will  
243 \* result in calling the parameterless method with name {@code methodName}).  
244 \* @param parameterTypes match these parameters - treat null as empty array  
245 \* @return The value returned by the invoked method  
246 \*  
247 \* @throws NoSuchMethodException if there is no such accessible method  
248 \* @throws InvocationTargetException wraps an exception thrown by the  
249 \* method invoked  
250 \* @throws IllegalAccessException if the requested method is not accessible  
251 \* via reflection  
252 \*/  
253 public static Object invokeMethod(  
254 final Object object,  
255 final String methodName,  
256 Object[] args,  
257 Class<?>[] parameterTypes)  
258 throws  
259 NoSuchMethodException,  
260 IllegalAccessException,  
261 InvocationTargetException {  
262  
263 if (parameterTypes == null) {  
264 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
265 }  
266 if (args == null) {  
267 args = EMPTY\_OBJECT\_ARRAY;  
268 }  
269  
270 final Method method = getMatchingAccessibleMethod(  
271 object.getClass(),  
272 methodName,  
273 parameterTypes);  
274 if (method == null) {  
275 throw new NoSuchMethodException("No such accessible method: " +  
276 methodName + "() on object: " + object.getClass().getName());  
277 }  
278 return method.invoke(object, args);  
279 }  
280  
281  
282 /\*\*  
283 \* <p>Invoke a method whose parameter type matches exactly the object  
284 \* type.</p>  
285 \*  
286 \* <p> This is a convenient wrapper for  
287 \* {@link #invokeExactMethod(Object object,String methodName,Object [] args)}.  
288 \* </p>  
289 \*  
290 \* @param object invoke method on this object  
291 \* @param methodName get method with this name  
292 \* @param arg use this argument. May be null (this will result in calling the  
293 \* parameterless method with name {@code methodName}).  
294 \* @return The value returned by the invoked method  
295 \*  
296 \* @throws NoSuchMethodException if there is no such accessible method  
297 \* @throws InvocationTargetException wraps an exception thrown by the  
298 \* method invoked  
299 \* @throws IllegalAccessException if the requested method is not accessible  
300 \* via reflection  
301 \*/  
302 public static Object invokeExactMethod(  
303 final Object object,  
304 final String methodName,  
305 final Object arg)  
306 throws  
307 NoSuchMethodException,  
308 IllegalAccessException,  
309 InvocationTargetException {  
310  
311 final Object[] args = toArray(arg);  
312 return invokeExactMethod(object, methodName, args);  
313 }  
314  
315  
316 /\*\*  
317 \* <p>Invoke a method whose parameter types match exactly the object  
318 \* types.</p>  
319 \*  
320 \* <p> This uses reflection to invoke the method obtained from a call to  
321 \* <code>getAccessibleMethod()</code>.</p>  
322 \*  
323 \* @param object invoke method on this object  
324 \* @param methodName get method with this name  
325 \* @param args use these arguments - treat null as empty array (passing null will  
326 \* result in calling the parameterless method with name {@code methodName}).  
327 \* @return The value returned by the invoked method  
328 \*  
329 \* @throws NoSuchMethodException if there is no such accessible method  
330 \* @throws InvocationTargetException wraps an exception thrown by the  
331 \* method invoked  
332 \* @throws IllegalAccessException if the requested method is not accessible  
333 \* via reflection  
334 \*/  
335 public static Object invokeExactMethod(  
336 final Object object,  
337 final String methodName,  
338 Object[] args)  
339 throws  
340 NoSuchMethodException,  
341 IllegalAccessException,  
342 InvocationTargetException {  
343  
344 if (args == null) {  
345 args = EMPTY\_OBJECT\_ARRAY;  
346 }  
347 final int arguments = args.length;  
348 final Class<?>[] parameterTypes = new Class[arguments];  
349 for (int i = 0; i < arguments; i++) {  
350 parameterTypes[i] = args[i].getClass();  
351 }  
352 return invokeExactMethod(object, methodName, args, parameterTypes);  
353 }  
354  
355  
356 /\*\*  
357 \* <p>Invoke a method whose parameter types match exactly the parameter  
358 \* types given.</p>  
359 \*  
360 \* <p>This uses reflection to invoke the method obtained from a call to  
361 \* <code>getAccessibleMethod()</code>.</p>  
362 \*  
363 \* @param object invoke method on this object  
364 \* @param methodName get method with this name  
365 \* @param args use these arguments - treat null as empty array (passing null will  
366 \* result in calling the parameterless method with name {@code methodName}).  
367 \* @param parameterTypes match these parameters - treat null as empty array  
368 \* @return The value returned by the invoked method  
369 \*  
370 \* @throws NoSuchMethodException if there is no such accessible method  
371 \* @throws InvocationTargetException wraps an exception thrown by the  
372 \* method invoked  
373 \* @throws IllegalAccessException if the requested method is not accessible  
374 \* via reflection  
375 \*/  
376 public static Object invokeExactMethod(  
377 final Object object,  
378 final String methodName,  
379 Object[] args,  
380 Class<?>[] parameterTypes)  
381 throws  
382 NoSuchMethodException,  
383 IllegalAccessException,  
384 InvocationTargetException {  
385  
386 if (args == null) {  
387 args = EMPTY\_OBJECT\_ARRAY;  
388 }  
389  
390 if (parameterTypes == null) {  
391 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
392 }  
393  
394 final Method method = getAccessibleMethod(  
395 object.getClass(),  
396 methodName,  
397 parameterTypes);  
398 if (method == null) {  
399 throw new NoSuchMethodException("No such accessible method: " +  
400 methodName + "() on object: " + object.getClass().getName());  
401 }  
402 return method.invoke(object, args);  
403 }  
404  
405 /\*\*  
406 \* <p>Invoke a static method whose parameter types match exactly the parameter  
407 \* types given.</p>  
408 \*  
409 \* <p>This uses reflection to invoke the method obtained from a call to  
410 \* {@link #getAccessibleMethod(Class, String, Class[])}.</p>  
411 \*  
412 \* @param objectClass invoke static method on this class  
413 \* @param methodName get method with this name  
414 \* @param args use these arguments - treat null as empty array (passing null will  
415 \* result in calling the parameterless method with name {@code methodName}).  
416 \* @param parameterTypes match these parameters - treat null as empty array  
417 \* @return The value returned by the invoked method  
418 \*  
419 \* @throws NoSuchMethodException if there is no such accessible method  
420 \* @throws InvocationTargetException wraps an exception thrown by the  
421 \* method invoked  
422 \* @throws IllegalAccessException if the requested method is not accessible  
423 \* via reflection  
424 \* @since 1.8.0  
425 \*/  
426 public static Object invokeExactStaticMethod(  
427 final Class<?> objectClass,  
428 final String methodName,  
429 Object[] args,  
430 Class<?>[] parameterTypes)  
431 throws  
432 NoSuchMethodException,  
433 IllegalAccessException,  
434 InvocationTargetException {  
435  
436 if (args == null) {  
437 args = EMPTY\_OBJECT\_ARRAY;  
438 }  
439  
440 if (parameterTypes == null) {  
441 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
442 }  
443  
444 final Method method = getAccessibleMethod(  
445 objectClass,  
446 methodName,  
447 parameterTypes);  
448 if (method == null) {  
449 throw new NoSuchMethodException("No such accessible method: " +  
450 methodName + "() on class: " + objectClass.getName());  
451 }  
452 return method.invoke(null, args);  
453 }  
454  
455 /\*\*  
456 \* <p>Invoke a named static method whose parameter type matches the object type.</p>  
457 \*  
458 \* <p>The behaviour of this method is less deterministic  
459 \* than {@link #invokeExactMethod(Object, String, Object[], Class[])}.  
460 \* It loops through all methods with names that match  
461 \* and then executes the first it finds with compatible parameters.</p>  
462 \*  
463 \* <p>This method supports calls to methods taking primitive parameters  
464 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
465 \* would match a <code>boolean</code> primitive.</p>  
466 \*  
467 \* <p> This is a convenient wrapper for  
468 \* {@link #invokeStaticMethod(Class objectClass,String methodName,Object [] args)}.  
469 \* </p>  
470 \*  
471 \* @param objectClass invoke static method on this class  
472 \* @param methodName get method with this name  
473 \* @param arg use this argument. May be null (this will result in calling the  
474 \* parameterless method with name {@code methodName}).  
475 \* @return The value returned by the invoked method  
476 \*  
477 \* @throws NoSuchMethodException if there is no such accessible method  
478 \* @throws InvocationTargetException wraps an exception thrown by the  
479 \* method invoked  
480 \* @throws IllegalAccessException if the requested method is not accessible  
481 \* via reflection  
482 \* @since 1.8.0  
483 \*/  
484 public static Object invokeStaticMethod(  
485 final Class<?> objectClass,  
486 final String methodName,  
487 final Object arg)  
488 throws  
489 NoSuchMethodException,  
490 IllegalAccessException,  
491 InvocationTargetException {  
492  
493 final Object[] args = toArray(arg);  
494 return invokeStaticMethod (objectClass, methodName, args);  
495 }  
496  
497  
498 /\*\*  
499 \* <p>Invoke a named static method whose parameter type matches the object type.</p>  
500 \*  
501 \* <p>The behaviour of this method is less deterministic  
502 \* than {@link #invokeExactMethod(Object object,String methodName,Object [] args)}.  
503 \* It loops through all methods with names that match  
504 \* and then executes the first it finds with compatible parameters.</p>  
505 \*  
506 \* <p>This method supports calls to methods taking primitive parameters  
507 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
508 \* would match a <code>boolean</code> primitive.</p>  
509 \*  
510 \* <p> This is a convenient wrapper for  
511 \* {@link #invokeStaticMethod(Class objectClass,String methodName,Object [] args,Class[] parameterTypes)}.  
512 \* </p>  
513 \*  
514 \* @param objectClass invoke static method on this class  
515 \* @param methodName get method with this name  
516 \* @param args use these arguments - treat null as empty array (passing null will  
517 \* result in calling the parameterless method with name {@code methodName}).  
518 \* @return The value returned by the invoked method  
519 \*  
520 \* @throws NoSuchMethodException if there is no such accessible method  
521 \* @throws InvocationTargetException wraps an exception thrown by the  
522 \* method invoked  
523 \* @throws IllegalAccessException if the requested method is not accessible  
524 \* via reflection  
525 \* @since 1.8.0  
526 \*/  
527 public static Object invokeStaticMethod(  
528 final Class<?> objectClass,  
529 final String methodName,  
530 Object[] args)  
531 throws  
532 NoSuchMethodException,  
533 IllegalAccessException,  
534 InvocationTargetException {  
535  
536 if (args == null) {  
537 args = EMPTY\_OBJECT\_ARRAY;  
538 }  
539 final int arguments = args.length;  
540 final Class<?>[] parameterTypes = new Class[arguments];  
541 for (int i = 0; i < arguments; i++) {  
542 parameterTypes[i] = args[i].getClass();  
543 }  
544 return invokeStaticMethod (objectClass, methodName, args, parameterTypes);  
545 }  
546  
547  
548 /\*\*  
549 \* <p>Invoke a named static method whose parameter type matches the object type.</p>  
550 \*  
551 \* <p>The behaviour of this method is less deterministic  
552 \* than {@link  
553 \* #invokeExactStaticMethod(Class objectClass,String methodName,Object [] args,Class[] parameterTypes)}.  
554 \* It loops through all methods with names that match  
555 \* and then executes the first it finds with compatible parameters.</p>  
556 \*  
557 \* <p>This method supports calls to methods taking primitive parameters  
558 \* via passing in wrapping classes. So, for example, a <code>Boolean</code> class  
559 \* would match a <code>boolean</code> primitive.</p>  
560 \*  
561 \*  
562 \* @param objectClass invoke static method on this class  
563 \* @param methodName get method with this name  
564 \* @param args use these arguments - treat null as empty array (passing null will  
565 \* result in calling the parameterless method with name {@code methodName}).  
566 \* @param parameterTypes match these parameters - treat null as empty array  
567 \* @return The value returned by the invoked method  
568 \*  
569 \* @throws NoSuchMethodException if there is no such accessible method  
570 \* @throws InvocationTargetException wraps an exception thrown by the  
571 \* method invoked  
572 \* @throws IllegalAccessException if the requested method is not accessible  
573 \* via reflection  
574 \* @since 1.8.0  
575 \*/  
576 public static Object invokeStaticMethod(  
577 final Class<?> objectClass,  
578 final String methodName,  
579 Object[] args,  
580 Class<?>[] parameterTypes)  
581 throws  
582 NoSuchMethodException,  
583 IllegalAccessException,  
584 InvocationTargetException {  
585  
586 if (parameterTypes == null) {  
587 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
588 }  
589 if (args == null) {  
590 args = EMPTY\_OBJECT\_ARRAY;  
591 }  
592  
593 final Method method = getMatchingAccessibleMethod(  
594 objectClass,  
595 methodName,  
596 parameterTypes);  
597 if (method == null) {  
598 throw new NoSuchMethodException("No such accessible method: " +  
599 methodName + "() on class: " + objectClass.getName());  
600 }  
601 return method.invoke(null, args);  
602 }  
603  
604  
605 /\*\*  
606 \* <p>Invoke a static method whose parameter type matches exactly the object  
607 \* type.</p>  
608 \*  
609 \* <p> This is a convenient wrapper for  
610 \* {@link #invokeExactStaticMethod(Class objectClass,String methodName,Object [] args)}.  
611 \* </p>  
612 \*  
613 \* @param objectClass invoke static method on this class  
614 \* @param methodName get method with this name  
615 \* @param arg use this argument. May be null (this will result in calling the  
616 \* parameterless method with name {@code methodName}).  
617 \* @return The value returned by the invoked method  
618 \*  
619 \* @throws NoSuchMethodException if there is no such accessible method  
620 \* @throws InvocationTargetException wraps an exception thrown by the  
621 \* method invoked  
622 \* @throws IllegalAccessException if the requested method is not accessible  
623 \* via reflection  
624 \* @since 1.8.0  
625 \*/  
626 public static Object invokeExactStaticMethod(  
627 final Class<?> objectClass,  
628 final String methodName,  
629 final Object arg)  
630 throws  
631 NoSuchMethodException,  
632 IllegalAccessException,  
633 InvocationTargetException {  
634  
635 final Object[] args = toArray(arg);  
636 return invokeExactStaticMethod (objectClass, methodName, args);  
637 }  
638  
639 /\*\*  
640 \* <p>Invoke a static method whose parameter types match exactly the object  
641 \* types.</p>  
642 \*  
643 \* <p> This uses reflection to invoke the method obtained from a call to  
644 \* {@link #getAccessibleMethod(Class, String, Class[])}.</p>  
645 \*  
646 \* @param objectClass invoke static method on this class  
647 \* @param methodName get method with this name  
648 \* @param args use these arguments - treat null as empty array (passing null will  
649 \* result in calling the parameterless method with name {@code methodName}).  
650 \* @return The value returned by the invoked method  
651 \*  
652 \* @throws NoSuchMethodException if there is no such accessible method  
653 \* @throws InvocationTargetException wraps an exception thrown by the  
654 \* method invoked  
655 \* @throws IllegalAccessException if the requested method is not accessible  
656 \* via reflection  
657 \* @since 1.8.0  
658 \*/  
659 public static Object invokeExactStaticMethod(  
660 final Class<?> objectClass,  
661 final String methodName,  
662 Object[] args)  
663 throws  
664 NoSuchMethodException,  
665 IllegalAccessException,  
666 InvocationTargetException {  
667  
668 if (args == null) {  
669 args = EMPTY\_OBJECT\_ARRAY;  
670 }  
671 final int arguments = args.length;  
672 final Class<?>[] parameterTypes = new Class[arguments];  
673 for (int i = 0; i < arguments; i++) {  
674 parameterTypes[i] = args[i].getClass();  
675 }  
676 return invokeExactStaticMethod(objectClass, methodName, args, parameterTypes);  
677 }  
678  
679  
680 private static Object[] toArray(final Object arg) {  
681 Object[] args = null;  
682 if (arg != null) {  
683 args = new Object[] { arg };  
684 }  
685 return args;  
686 }  
687  
688 /\*\*  
689 \* <p>Return an accessible method (that is, one that can be invoked via  
690 \* reflection) with given name and a single parameter. If no such method  
691 \* can be found, return <code>null</code>.  
692 \* Basically, a convenience wrapper that constructs a <code>Class</code>  
693 \* array for you.</p>  
694 \*  
695 \* @param clazz get method from this class  
696 \* @param methodName get method with this name  
697 \* @param parameterType taking this type of parameter  
698 \* @return The accessible method  
699 \*/  
700 public static Method getAccessibleMethod(  
701 final Class<?> clazz,  
702 final String methodName,  
703 final Class<?> parameterType) {  
704  
705 final Class<?>[] parameterTypes = {parameterType};  
706 return getAccessibleMethod(clazz, methodName, parameterTypes);  
707 }  
708  
709  
710 /\*\*  
711 \* <p>Return an accessible method (that is, one that can be invoked via  
712 \* reflection) with given name and parameters. If no such method  
713 \* can be found, return <code>null</code>.  
714 \* This is just a convenient wrapper for  
715 \* {@link #getAccessibleMethod(Method method)}.</p>  
716 \*  
717 \* @param clazz get method from this class  
718 \* @param methodName get method with this name  
719 \* @param parameterTypes with these parameters types  
720 \* @return The accessible method  
721 \*/  
722 public static Method getAccessibleMethod(  
723 final Class<?> clazz,  
724 final String methodName,  
725 final Class<?>[] parameterTypes) {  
726  
727 try {  
728 final MethodDescriptor md = new MethodDescriptor(clazz, methodName, parameterTypes, true);  
729 // Check the cache first  
730 Method method = getCachedMethod(md);  
731 if (method != null) {  
732 return method;  
733 }  
734  
735 method = getAccessibleMethod  
736 (clazz, clazz.getMethod(methodName, parameterTypes));  
737 cacheMethod(md, method);  
738 return method;  
739 } catch (final NoSuchMethodException e) {  
740 return (null);  
741 }  
742 }  
743  
744 /\*\*  
745 \* <p>Return an accessible method (that is, one that can be invoked via  
746 \* reflection) that implements the specified Method. If no such method  
747 \* can be found, return <code>null</code>.</p>  
748 \*  
749 \* @param method The method that we wish to call  
750 \* @return The accessible method  
751 \*/  
752 public static Method getAccessibleMethod(final Method method) {  
753  
754 // Make sure we have a method to check  
755 if (method == null) {  
756 return (null);  
757 }  
758  
759 return getAccessibleMethod(method.getDeclaringClass(), method);  
760 }  
761  
762  
763  
764 /\*\*  
765 \* <p>Return an accessible method (that is, one that can be invoked via  
766 \* reflection) that implements the specified Method. If no such method  
767 \* can be found, return <code>null</code>.</p>  
768 \*  
769 \* @param clazz The class of the object  
770 \* @param method The method that we wish to call  
771 \* @return The accessible method  
772 \* @since 1.8.0  
773 \*/  
774 public static Method getAccessibleMethod(Class<?> clazz, Method method) {  
775  
776 // Make sure we have a method to check  
777 if (method == null) {  
778 return (null);  
779 }  
780  
781 // If the requested method is not public we cannot call it  
782 if (!Modifier.isPublic(method.getModifiers())) {  
783 return (null);  
784 }  
785  
786 boolean sameClass = true;  
787 if (clazz == null) {  
788 clazz = method.getDeclaringClass();  
789 } else {  
790 sameClass = clazz.equals(method.getDeclaringClass());  
791 if (!method.getDeclaringClass().isAssignableFrom(clazz)) {  
792 throw new IllegalArgumentException(clazz.getName() +  
793 " is not assignable from " + method.getDeclaringClass().getName());  
794 }  
795 }  
796  
797 // If the class is public, we are done  
798 if (Modifier.isPublic(clazz.getModifiers())) {  
799 if (!sameClass && !Modifier.isPublic(method.getDeclaringClass().getModifiers())) {  
800 setMethodAccessible(method); // Default access superclass workaround  
801 }  
802 return (method);  
803 }  
804  
805 final String methodName = method.getName();  
806 final Class<?>[] parameterTypes = method.getParameterTypes();  
807  
808 // Check the implemented interfaces and subinterfaces  
809 method =  
810 getAccessibleMethodFromInterfaceNest(clazz,  
811 methodName,  
812 parameterTypes);  
813  
814 // Check the superclass chain  
815 if (method == null) {  
816 method = getAccessibleMethodFromSuperclass(clazz,  
817 methodName,  
818 parameterTypes);  
819 }  
820  
821 return (method);  
822 }  
823  
824  
825 // -------------------------------------------------------- Private Methods  
826  
827 /\*\*  
828 \* <p>Return an accessible method (that is, one that can be invoked via  
829 \* reflection) by scanning through the superclasses. If no such method  
830 \* can be found, return <code>null</code>.</p>  
831 \*  
832 \* @param clazz Class to be checked  
833 \* @param methodName Method name of the method we wish to call  
834 \* @param parameterTypes The parameter type signatures  
835 \*/  
836 private static Method getAccessibleMethodFromSuperclass  
837 (final Class<?> clazz, final String methodName, final Class<?>[] parameterTypes) {  
838  
839 Class<?> parentClazz = clazz.getSuperclass();  
840 while (parentClazz != null) {  
841 if (Modifier.isPublic(parentClazz.getModifiers())) {  
842 try {  
843 return parentClazz.getMethod(methodName, parameterTypes);  
844 } catch (final NoSuchMethodException e) {  
845 return null;  
846 }  
847 }  
848 parentClazz = parentClazz.getSuperclass();  
849 }  
850 return null;  
851 }  
852  
853 /\*\*  
854 \* <p>Return an accessible method (that is, one that can be invoked via  
855 \* reflection) that implements the specified method, by scanning through  
856 \* all implemented interfaces and subinterfaces. If no such method  
857 \* can be found, return <code>null</code>.</p>  
858 \*  
859 \* <p> There isn't any good reason why this method must be private.  
860 \* It is because there doesn't seem any reason why other classes should  
861 \* call this rather than the higher level methods.</p>  
862 \*  
863 \* @param clazz Parent class for the interfaces to be checked  
864 \* @param methodName Method name of the method we wish to call  
865 \* @param parameterTypes The parameter type signatures  
866 \*/  
867 private static Method getAccessibleMethodFromInterfaceNest  
868 (Class<?> clazz, final String methodName, final Class<?>[] parameterTypes) {  
869  
870 Method method = null;  
871  
872 // Search up the superclass chain  
873 for (; clazz != null; clazz = clazz.getSuperclass()) {  
874  
875 // Check the implemented interfaces of the parent class  
876 final Class<?>[] interfaces = clazz.getInterfaces();  
877 for (int i = 0; i < interfaces.length; i++) {  
878  
879 // Is this interface public?  
880 if (!Modifier.isPublic(interfaces[i].getModifiers())) {  
881 continue;  
882 }  
883  
884 // Does the method exist on this interface?  
885 try {  
886 method = interfaces[i].getDeclaredMethod(methodName,  
887 parameterTypes);  
888 } catch (final NoSuchMethodException e) {  
889 /\* Swallow, if no method is found after the loop then this  
890 \* method returns null.  
891 \*/  
892 }  
893 if (method != null) {  
894 return method;  
895 }  
896  
897 // Recursively check our parent interfaces  
898 method =  
899 getAccessibleMethodFromInterfaceNest(interfaces[i],  
900 methodName,  
901 parameterTypes);  
902 if (method != null) {  
903 return method;  
904 }  
905  
906 }  
907  
908 }  
909  
910 // We did not find anything  
911 return (null);  
912 }  
913  
914 /\*\*  
915 \* <p>Find an accessible method that matches the given name and has compatible parameters.  
916 \* Compatible parameters mean that every method parameter is assignable from  
917 \* the given parameters.  
918 \* In other words, it finds a method with the given name  
919 \* that will take the parameters given.<p>  
920 \*  
921 \* <p>This method is slightly undeterministic since it loops  
922 \* through methods names and return the first matching method.</p>  
923 \*  
924 \* <p>This method is used by  
925 \* {@link  
926 \* #invokeMethod(Object object,String methodName,Object [] args,Class[] parameterTypes)}.  
927 \*  
928 \* <p>This method can match primitive parameter by passing in wrapper classes.  
929 \* For example, a <code>Boolean</code> will match a primitive <code>boolean</code>  
930 \* parameter.  
931 \*  
932 \* @param clazz find method in this class  
933 \* @param methodName find method with this name  
934 \* @param parameterTypes find method with compatible parameters  
935 \* @return The accessible method  
936 \*/  
937 public static Method getMatchingAccessibleMethod(  
938 final Class<?> clazz,  
939 final String methodName,  
940 final Class<?>[] parameterTypes) {  
941 // trace logging  
942 final Log log = LogFactory.getLog(MethodUtils.class);  
943 if (log.isTraceEnabled()) {  
944 log.trace("Matching name=" + methodName + " on " + clazz);  
945 }  
946 final MethodDescriptor md = new MethodDescriptor(clazz, methodName, parameterTypes, false);  
947  
948 // see if we can find the method directly  
949 // most of the time this works and it's much faster  
950 try {  
951 // Check the cache first  
952 Method method = getCachedMethod(md);  
953 if (method != null) {  
954 return method;  
955 }  
956  
957 method = clazz.getMethod(methodName, parameterTypes);  
958 if (log.isTraceEnabled()) {  
959 log.trace("Found straight match: " + method);  
960 log.trace("isPublic:" + Modifier.isPublic(method.getModifiers()));  
961 }  
962  
963 setMethodAccessible(method); // Default access superclass workaround  
964  
965 cacheMethod(md, method);  
966 return method;  
967  
968 } catch (final NoSuchMethodException e) { /\* SWALLOW \*/ }  
969  
970 // search through all methods  
971 final int paramSize = parameterTypes.length;  
972 Method bestMatch = null;  
973 final Method[] methods = clazz.getMethods();  
974 float bestMatchCost = Float.MAX\_VALUE;  
975 float myCost = Float.MAX\_VALUE;  
976 for (Method method2 : methods) {  
977 if (method2.getName().equals(methodName)) {  
978 // log some trace information  
979 if (log.isTraceEnabled()) {  
980 log.trace("Found matching name:");  
981 log.trace(method2);  
982 }  
983  
984 // compare parameters  
985 final Class<?>[] methodsParams = method2.getParameterTypes();  
986 final int methodParamSize = methodsParams.length;  
987 if (methodParamSize == paramSize) {  
988 boolean match = true;  
989 for (int n = 0 ; n < methodParamSize; n++) {  
990 if (log.isTraceEnabled()) {  
991 log.trace("Param=" + parameterTypes[n].getName());  
992 log.trace("Method=" + methodsParams[n].getName());  
993 }  
994 if (!isAssignmentCompatible(methodsParams[n], parameterTypes[n])) {  
995 if (log.isTraceEnabled()) {  
996 log.trace(methodsParams[n] + " is not assignable from "  
997 + parameterTypes[n]);  
998 }  
999 match = false;  
1000 break;  
1001 }  
1002 }  
1003  
1004 if (match) {  
1005 // get accessible version of method  
1006 final Method method = getAccessibleMethod(clazz, method2);  
1007 if (method != null) {  
1008 if (log.isTraceEnabled()) {  
1009 log.trace(method + " accessible version of "  
1010 + method2);  
1011 }  
1012 setMethodAccessible(method); // Default access superclass workaround  
1013 myCost = getTotalTransformationCost(parameterTypes,method.getParameterTypes());  
1014 if ( myCost < bestMatchCost ) {  
1015 bestMatch = method;  
1016 bestMatchCost = myCost;  
1017 }  
1018 }  
1019  
1020 log.trace("Couldn't find accessible method.");  
1021 }  
1022 }  
1023 }  
1024 }  
1025 if ( bestMatch != null ){  
1026 cacheMethod(md, bestMatch);  
1027 } else {  
1028 // didn't find a match  
1029 log.trace("No match found.");  
1030 }  
1031  
1032 return bestMatch;  
1033 }  
1034  
1035 /\*\*  
1036 \* Try to make the method accessible  
1037 \* @param method The source arguments  
1038 \*/  
1039 private static void setMethodAccessible(final Method method) {  
1040 try {  
1041 //  
1042 // XXX Default access superclass workaround  
1043 //  
1044 // When a public class has a default access superclass  
1045 // with public methods, these methods are accessible.  
1046 // Calling them from compiled code works fine.  
1047 //  
1048 // Unfortunately, using reflection to invoke these methods  
1049 // seems to (wrongly) to prevent access even when the method  
1050 // modifer is public.  
1051 //  
1052 // The following workaround solves the problem but will only  
1053 // work from sufficiently privilages code.  
1054 //  
1055 // Better workarounds would be greatfully accepted.  
1056 //  
1057 if (!method.isAccessible()) {  
1058 method.setAccessible(true);  
1059 }  
1060  
1061 } catch (final SecurityException se) {  
1062 // log but continue just in case the method.invoke works anyway  
1063 final Log log = LogFactory.getLog(MethodUtils.class);  
1064 if (!loggedAccessibleWarning) {  
1065 boolean vulnerableJVM = false;  
1066 try {  
1067 final String specVersion = System.getProperty("java.specification.version");  
1068 if (specVersion.charAt(0) == '1' &&  
1069 (specVersion.charAt(2) == '0' ||  
1070 specVersion.charAt(2) == '1' ||  
1071 specVersion.charAt(2) == '2' ||  
1072 specVersion.charAt(2) == '3')) {  
1073  
1074 vulnerableJVM = true;  
1075 }  
1076 } catch (final SecurityException e) {  
1077 // don't know - so display warning  
1078 vulnerableJVM = true;  
1079 }  
1080 if (vulnerableJVM) {  
1081 log.warn(  
1082 "Current Security Manager restricts use of workarounds for reflection bugs "  
1083 + " in pre-1.4 JVMs.");  
1084 }  
1085 loggedAccessibleWarning = true;  
1086 }  
1087 log.debug("Cannot setAccessible on method. Therefore cannot use jvm access bug workaround.", se);  
1088 }  
1089 }  
1090  
1091 /\*\*  
1092 \* Returns the sum of the object transformation cost for each class in the source  
1093 \* argument list.  
1094 \* @param srcArgs The source arguments  
1095 \* @param destArgs The destination arguments  
1096 \* @return The total transformation cost  
1097 \*/  
1098 private static float getTotalTransformationCost(final Class<?>[] srcArgs, final Class<?>[] destArgs) {  
1099  
1100 float totalCost = 0.0f;  
1101 for (int i = 0; i < srcArgs.length; i++) {  
1102 Class<?> srcClass, destClass;  
1103 srcClass = srcArgs[i];  
1104 destClass = destArgs[i];  
1105 totalCost += getObjectTransformationCost(srcClass, destClass);  
1106 }  
1107  
1108 return totalCost;  
1109 }  
1110  
1111 /\*\*  
1112 \* Gets the number of steps required needed to turn the source class into the  
1113 \* destination class. This represents the number of steps in the object hierarchy  
1114 \* graph.  
1115 \* @param srcClass The source class  
1116 \* @param destClass The destination class  
1117 \* @return The cost of transforming an object  
1118 \*/  
1119 private static float getObjectTransformationCost(Class<?> srcClass, final Class<?> destClass) {  
1120 float cost = 0.0f;  
1121 while (srcClass != null && !destClass.equals(srcClass)) {  
1122 if (destClass.isPrimitive()) {  
1123 final Class<?> destClassWrapperClazz = getPrimitiveWrapper(destClass);  
1124 if (destClassWrapperClazz != null && destClassWrapperClazz.equals(srcClass)) {  
1125 cost += 0.25f;  
1126 break;  
1127 }  
1128 }  
1129 if (destClass.isInterface() && isAssignmentCompatible(destClass,srcClass)) {  
1130 // slight penalty for interface match.  
1131 // we still want an exact match to override an interface match, but  
1132 // an interface match should override anything where we have to get a  
1133 // superclass.  
1134 cost += 0.25f;  
1135 break;  
1136 }  
1137 cost++;  
1138 srcClass = srcClass.getSuperclass();  
1139 }  
1140  
1141 /\*  
1142 \* If the destination class is null, we've travelled all the way up to  
1143 \* an Object match. We'll penalize this by adding 1.5 to the cost.  
1144 \*/  
1145 if (srcClass == null) {  
1146 cost += 1.5f;  
1147 }  
1148  
1149 return cost;  
1150 }  
1151  
1152  
1153 /\*\*  
1154 \* <p>Determine whether a type can be used as a parameter in a method invocation.  
1155 \* This method handles primitive conversions correctly.</p>  
1156 \*  
1157 \* <p>In order words, it will match a <code>Boolean</code> to a <code>boolean</code>,  
1158 \* a <code>Long</code> to a <code>long</code>,  
1159 \* a <code>Float</code> to a <code>float</code>,  
1160 \* a <code>Integer</code> to a <code>int</code>,  
1161 \* and a <code>Double</code> to a <code>double</code>.  
1162 \* Now logic widening matches are allowed.  
1163 \* For example, a <code>Long</code> will not match a <code>int</code>.  
1164 \*  
1165 \* @param parameterType the type of parameter accepted by the method  
1166 \* @param parameterization the type of parameter being tested  
1167 \*  
1168 \* @return true if the assignment is compatible.  
1169 \*/  
1170 public static final boolean isAssignmentCompatible(final Class<?> parameterType, final Class<?> parameterization) {  
1171 // try plain assignment  
1172 if (parameterType.isAssignableFrom(parameterization)) {  
1173 return true;  
1174 }  
1175  
1176 if (parameterType.isPrimitive()) {  
1177 // this method does \*not\* do widening - you must specify exactly  
1178 // is this the right behaviour?  
1179 final Class<?> parameterWrapperClazz = getPrimitiveWrapper(parameterType);  
1180 if (parameterWrapperClazz != null) {  
1181 return parameterWrapperClazz.equals(parameterization);  
1182 }  
1183 }  
1184  
1185 return false;  
1186 }  
1187  
1188 /\*\*  
1189 \* Gets the wrapper object class for the given primitive type class.  
1190 \* For example, passing <code>boolean.class</code> returns <code>Boolean.class</code>  
1191 \* @param primitiveType the primitive type class for which a match is to be found  
1192 \* @return the wrapper type associated with the given primitive  
1193 \* or null if no match is found  
1194 \*/  
1195 public static Class<?> getPrimitiveWrapper(final Class<?> primitiveType) {  
1196 // does anyone know a better strategy than comparing names?  
1197 if (boolean.class.equals(primitiveType)) {  
1198 return Boolean.class;  
1199 } else if (float.class.equals(primitiveType)) {  
1200 return Float.class;  
1201 } else if (long.class.equals(primitiveType)) {  
1202 return Long.class;  
1203 } else if (int.class.equals(primitiveType)) {  
1204 return Integer.class;  
1205 } else if (short.class.equals(primitiveType)) {  
1206 return Short.class;  
1207 } else if (byte.class.equals(primitiveType)) {  
1208 return Byte.class;  
1209 } else if (double.class.equals(primitiveType)) {  
1210 return Double.class;  
1211 } else if (char.class.equals(primitiveType)) {  
1212 return Character.class;  
1213 } else {  
1214  
1215 return null;  
1216 }  
1217 }  
1218  
1219 /\*\*  
1220 \* Gets the class for the primitive type corresponding to the primitive wrapper class given.  
1221 \* For example, an instance of <code>Boolean.class</code> returns a <code>boolean.class</code>.  
1222 \* @param wrapperType the  
1223 \* @return the primitive type class corresponding to the given wrapper class,  
1224 \* null if no match is found  
1225 \*/  
1226 public static Class<?> getPrimitiveType(final Class<?> wrapperType) {  
1227 // does anyone know a better strategy than comparing names?  
1228 if (Boolean.class.equals(wrapperType)) {  
1229 return boolean.class;  
1230 } else if (Float.class.equals(wrapperType)) {  
1231 return float.class;  
1232 } else if (Long.class.equals(wrapperType)) {  
1233 return long.class;  
1234 } else if (Integer.class.equals(wrapperType)) {  
1235 return int.class;  
1236 } else if (Short.class.equals(wrapperType)) {  
1237 return short.class;  
1238 } else if (Byte.class.equals(wrapperType)) {  
1239 return byte.class;  
1240 } else if (Double.class.equals(wrapperType)) {  
1241 return double.class;  
1242 } else if (Character.class.equals(wrapperType)) {  
1243 return char.class;  
1244 } else {  
1245 final Log log = LogFactory.getLog(MethodUtils.class);  
1246 if (log.isDebugEnabled()) {  
1247 log.debug("Not a known primitive wrapper class: " + wrapperType);  
1248 }  
1249 return null;  
1250 }  
1251 }  
1252  
1253 /\*\*  
1254 \* Find a non primitive representation for given primitive class.  
1255 \*  
1256 \* @param clazz the class to find a representation for, not null  
1257 \* @return the original class if it not a primitive. Otherwise the wrapper class. Not null  
1258 \*/  
1259 public static Class<?> toNonPrimitiveClass(final Class<?> clazz) {  
1260 if (clazz.isPrimitive()) {  
1261 final Class<?> primitiveClazz = MethodUtils.getPrimitiveWrapper(clazz);  
1262 // the above method returns  
1263 if (primitiveClazz != null) {  
1264 return primitiveClazz;  
1265 } else {  
1266 return clazz;  
1267 }  
1268 } else {  
1269 return clazz;  
1270 }  
1271 }  
1272  
1273  
1274 /\*\*  
1275 \* Return the method from the cache, if present.  
1276 \*  
1277 \* @param md The method descriptor  
1278 \* @return The cached method  
1279 \*/  
1280 private static Method getCachedMethod(final MethodDescriptor md) {  
1281 if (CACHE\_METHODS) {  
1282 final Reference<Method> methodRef = cache.get(md);  
1283 if (methodRef != null) {  
1284 return methodRef.get();  
1285 }  
1286 }  
1287 return null;  
1288 }  
1289  
1290 /\*\*  
1291 \* Add a method to the cache.  
1292 \*  
1293 \* @param md The method descriptor  
1294 \* @param method The method to cache  
1295 \*/  
1296 private static void cacheMethod(final MethodDescriptor md, final Method method) {  
1297 if (CACHE\_METHODS) {  
1298 if (method != null) {  
1299 cache.put(md, new WeakReference<Method>(method));  
1300 }  
1301 }  
1302 }  
1303  
1304 /\*\*  
1305 \* Represents the key to looking up a Method by reflection.  
1306 \*/  
1307 private static class MethodDescriptor {  
1308 private final Class<?> cls;  
1309 private final String methodName;  
1310 private final Class<?>[] paramTypes;  
1311 private final boolean exact;  
1312 private final int hashCode;  
1313  
1314 /\*\*  
1315 \* The sole constructor.  
1316 \*  
1317 \* @param cls the class to reflect, must not be null  
1318 \* @param methodName the method name to obtain  
1319 \* @param paramTypes the array of classes representing the parameter types  
1320 \* @param exact whether the match has to be exact.  
1321 \*/  
1322 public MethodDescriptor(final Class<?> cls, final String methodName, Class<?>[] paramTypes, final boolean exact) {  
1323 if (cls == null) {  
1324 throw new IllegalArgumentException("Class cannot be null");  
1325 }  
1326 if (methodName == null) {  
1327 throw new IllegalArgumentException("Method Name cannot be null");  
1328 }  
1329 if (paramTypes == null) {  
1330 paramTypes = EMPTY\_CLASS\_PARAMETERS;  
1331 }  
1332  
1333 this.cls = cls;  
1334 this.methodName = methodName;  
1335 this.paramTypes = paramTypes;  
1336 this.exact= exact;  
1337  
1338 this.hashCode = methodName.length();  
1339 }  
1340 /\*\*  
1341 \* Checks for equality.  
1342 \* @param obj object to be tested for equality  
1343 \* @return true, if the object describes the same Method.  
1344 \*/  
1345 @Override  
1346 public boolean equals(final Object obj) {  
1347 if (!(obj instanceof MethodDescriptor)) {  
1348 return false;  
1349 }  
1350 final MethodDescriptor md = (MethodDescriptor)obj;  
1351  
1352 return (  
1353 exact == md.exact &&  
1354 methodName.equals(md.methodName) &&  
1355 cls.equals(md.cls) &&  
1356 java.util.Arrays.equals(paramTypes, md.paramTypes)  
1357 );  
1358 }  
1359 /\*\*  
1360 \* Returns the string length of method name. I.e. if the  
1361 \* hashcodes are different, the objects are different. If the  
1362 \* hashcodes are the same, need to use the equals method to  
1363 \* determine equality.  
1364 \* @return the string length of method name.  
1365 \*/  
1366 @Override  
1367 public int hashCode() {  
1368 return hashCode;  
1369 }  
1370 }  
1371}