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  
020import java.lang.reflect.Constructor;  
021import java.lang.reflect.InvocationTargetException;  
022import java.lang.reflect.Modifier;  
023  
024/\*\*  
025 \* <p> Utility reflection methods focussed on constructors, modelled after {@link MethodUtils}. </p>  
026 \*  
027 \* <h3>Known Limitations</h3>  
028 \* <h4>Accessing Public Constructors In A Default Access Superclass</h4>  
029 \* <p>There is an issue when invoking public constructors contained in a default access superclass.  
030 \* Reflection locates these constructors fine and correctly assigns them as public.  
031 \* However, an <code>IllegalAccessException</code> is thrown if the constructors is invoked.</p>  
032 \*  
033 \* <p><code>ConstructorUtils</code> contains a workaround for this situation.  
034 \* It will attempt to call <code>setAccessible</code> on this constructor.  
035 \* If this call succeeds, then the method can be invoked as normal.  
036 \* This call will only succeed when the application has sufficient security privilages.  
037 \* If this call fails then a warning will be logged and the method may fail.</p>  
038 \*  
039 \* @version $Id$  
040 \*/  
041public class ConstructorUtils {  
042  
043 // --------------------------------------------------------- Private Members  
044 /\*\* An empty class array \*/  
045 private static final Class<?>[] EMPTY\_CLASS\_PARAMETERS = new Class<?>[0];  
046 /\*\* An empty object array \*/  
047 private static final Object[] EMPTY\_OBJECT\_ARRAY = new Object[0];  
048  
049 // --------------------------------------------------------- Public Methods  
050  
051 /\*\*  
052 \* <p>Convenience method returning new instance of <code>klazz</code> using a single argument constructor.  
053 \* The formal parameter type is inferred from the actual values of <code>arg</code>.  
054 \* See {@link #invokeExactConstructor(Class, Object[], Class[])} for more details.</p>  
055 \*  
056 \* <p>The signatures should be assignment compatible.</p>  
057 \*  
058 \* @param <T> the type of the object to be constructed  
059 \* @param klass the class to be constructed.  
060 \* @param arg the actual argument. May be null (this will result in calling the default constructor).  
061 \* @return new instance of <code>klazz</code>  
062 \*  
063 \* @throws NoSuchMethodException If the constructor cannot be found  
064 \* @throws IllegalAccessException If an error occurs accessing the constructor  
065 \* @throws InvocationTargetException If an error occurs invoking the constructor  
066 \* @throws InstantiationException If an error occurs instantiating the class  
067 \*  
068 \* @see #invokeConstructor(java.lang.Class, java.lang.Object[], java.lang.Class[])  
069 \*/  
070 public static <T> T invokeConstructor(final Class<T> klass, final Object arg)  
071 throws  
072 NoSuchMethodException,  
073 IllegalAccessException,  
074 InvocationTargetException,  
075 InstantiationException {  
076  
077 final Object[] args = toArray(arg);  
078 return invokeConstructor(klass, args);  
079 }  
080  
081 /\*\*  
082 \* <p>Returns new instance of <code>klazz</code> created using the actual arguments <code>args</code>.  
083 \* The formal parameter types are inferred from the actual values of <code>args</code>.  
084 \* See {@link #invokeExactConstructor(Class, Object[], Class[])} for more details.</p>  
085 \*  
086 \* <p>The signatures should be assignment compatible.</p>  
087 \*  
088 \* @param <T> the type of the object to be constructed  
089 \* @param klass the class to be constructed.  
090 \* @param args actual argument array. May be null (this will result in calling the default constructor).  
091 \* @return new instance of <code>klazz</code>  
092 \*  
093 \* @throws NoSuchMethodException If the constructor cannot be found  
094 \* @throws IllegalAccessException If an error occurs accessing the constructor  
095 \* @throws InvocationTargetException If an error occurs invoking the constructor  
096 \* @throws InstantiationException If an error occurs instantiating the class  
097 \*  
098 \* @see #invokeConstructor(java.lang.Class, java.lang.Object[], java.lang.Class[])  
099 \*/  
100 public static <T> T invokeConstructor(final Class<T> klass, Object[] args)  
101 throws  
102 NoSuchMethodException,  
103 IllegalAccessException,  
104 InvocationTargetException,  
105 InstantiationException {  
106  
107 if (null == args) {  
108 args = EMPTY\_OBJECT\_ARRAY;  
109 }  
110 final int arguments = args.length;  
111 final Class<?> parameterTypes[] = new Class<?>[arguments];  
112 for (int i = 0; i < arguments; i++) {  
113 parameterTypes[i] = args[i].getClass();  
114 }  
115 return invokeConstructor(klass, args, parameterTypes);  
116 }  
117  
118 /\*\*  
119 \* <p>Returns new instance of <code>klazz</code> created using constructor  
120 \* with signature <code>parameterTypes</code> and actual arguments <code>args</code>.</p>  
121 \*  
122 \* <p>The signatures should be assignment compatible.</p>  
123 \*  
124 \* @param <T> the type of the object to be constructed  
125 \* @param klass the class to be constructed.  
126 \* @param args actual argument array. May be null (this will result in calling the default constructor).  
127 \* @param parameterTypes parameter types array  
128 \* @return new instance of <code>klazz</code>  
129 \*  
130 \* @throws NoSuchMethodException if matching constructor cannot be found  
131 \* @throws IllegalAccessException thrown on the constructor's invocation  
132 \* @throws InvocationTargetException thrown on the constructor's invocation  
133 \* @throws InstantiationException thrown on the constructor's invocation  
134 \* @see Constructor#newInstance  
135 \*/  
136 public static <T> T invokeConstructor(  
137 final Class<T> klass,  
138 Object[] args,  
139 Class<?>[] parameterTypes)  
140 throws  
141 NoSuchMethodException,  
142 IllegalAccessException,  
143 InvocationTargetException,  
144 InstantiationException {  
145  
146 if (parameterTypes == null) {  
147 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
148 }  
149 if (args == null) {  
150 args = EMPTY\_OBJECT\_ARRAY;  
151 }  
152  
153 final Constructor<T> ctor =  
154 getMatchingAccessibleConstructor(klass, parameterTypes);  
155 if (null == ctor) {  
156 throw new NoSuchMethodException(  
157 "No such accessible constructor on object: " + klass.getName());  
158 }  
159 return ctor.newInstance(args);  
160 }  
161  
162  
163 /\*\*  
164 \* <p>Convenience method returning new instance of <code>klazz</code> using a single argument constructor.  
165 \* The formal parameter type is inferred from the actual values of <code>arg</code>.  
166 \* See {@link #invokeExactConstructor(Class, Object[], Class[])} for more details.</p>  
167 \*  
168 \* <p>The signatures should match exactly.</p>  
169 \*  
170 \* @param <T> the type of the object to be constructed  
171 \* @param klass the class to be constructed.  
172 \* @param arg the actual argument. May be null (this will result in calling the default constructor).  
173 \* @return new instance of <code>klazz</code>  
174 \*  
175 \* @throws NoSuchMethodException If the constructor cannot be found  
176 \* @throws IllegalAccessException If an error occurs accessing the constructor  
177 \* @throws InvocationTargetException If an error occurs invoking the constructor  
178 \* @throws InstantiationException If an error occurs instantiating the class  
179 \*  
180 \* @see #invokeExactConstructor(java.lang.Class, java.lang.Object[], java.lang.Class[])  
181 \*/  
182 public static <T> T invokeExactConstructor(final Class<T> klass, final Object arg)  
183 throws  
184 NoSuchMethodException,  
185 IllegalAccessException,  
186 InvocationTargetException,  
187 InstantiationException {  
188  
189 final Object[] args = toArray(arg);  
190 return invokeExactConstructor(klass, args);  
191 }  
192  
193 /\*\*  
194 \* <p>Returns new instance of <code>klazz</code> created using the actual arguments <code>args</code>.  
195 \* The formal parameter types are inferred from the actual values of <code>args</code>.  
196 \* See {@link #invokeExactConstructor(Class, Object[], Class[])} for more details.</p>  
197 \*  
198 \* <p>The signatures should match exactly.</p>  
199 \*  
200 \* @param <T> the type of the object to be constructed  
201 \* @param klass the class to be constructed.  
202 \* @param args actual argument array. May be null (this will result in calling the default constructor).  
203 \* @return new instance of <code>klazz</code>  
204 \*  
205 \* @throws NoSuchMethodException If the constructor cannot be found  
206 \* @throws IllegalAccessException If an error occurs accessing the constructor  
207 \* @throws InvocationTargetException If an error occurs invoking the constructor  
208 \* @throws InstantiationException If an error occurs instantiating the class  
209 \*  
210 \* @see #invokeExactConstructor(java.lang.Class, java.lang.Object[], java.lang.Class[])  
211 \*/  
212 public static <T> T invokeExactConstructor(final Class<T> klass, Object[] args)  
213 throws  
214 NoSuchMethodException,  
215 IllegalAccessException,  
216 InvocationTargetException,  
217 InstantiationException {  
218  
219 if (null == args) {  
220 args = EMPTY\_OBJECT\_ARRAY;  
221 }  
222 final int arguments = args.length;  
223 final Class<?> parameterTypes[] = new Class[arguments];  
224 for (int i = 0; i < arguments; i++) {  
225 parameterTypes[i] = args[i].getClass();  
226 }  
227 return invokeExactConstructor(klass, args, parameterTypes);  
228 }  
229  
230 /\*\*  
231 \* <p>Returns new instance of <code>klazz</code> created using constructor  
232 \* with signature <code>parameterTypes</code> and actual arguments  
233 \* <code>args</code>.</p>  
234 \*  
235 \* <p>The signatures should match exactly.</p>  
236 \*  
237 \* @param <T> the type of the object to be constructed  
238 \* @param klass the class to be constructed.  
239 \* @param args actual argument array. May be null (this will result in calling the default constructor).  
240 \* @param parameterTypes parameter types array  
241 \* @return new instance of <code>klazz</code>  
242 \*  
243 \* @throws NoSuchMethodException if matching constructor cannot be found  
244 \* @throws IllegalAccessException thrown on the constructor's invocation  
245 \* @throws InvocationTargetException thrown on the constructor's invocation  
246 \* @throws InstantiationException thrown on the constructor's invocation  
247 \* @see Constructor#newInstance  
248 \*/  
249 public static <T> T invokeExactConstructor(  
250 final Class<T> klass,  
251 Object[] args,  
252 Class<?>[] parameterTypes)  
253 throws  
254 NoSuchMethodException,  
255 IllegalAccessException,  
256 InvocationTargetException,  
257 InstantiationException {  
258  
259 if (args == null) {  
260 args = EMPTY\_OBJECT\_ARRAY;  
261 }  
262  
263 if (parameterTypes == null) {  
264 parameterTypes = EMPTY\_CLASS\_PARAMETERS;  
265 }  
266  
267 final Constructor<T> ctor = getAccessibleConstructor(klass, parameterTypes);  
268 if (null == ctor) {  
269 throw new NoSuchMethodException(  
270 "No such accessible constructor on object: " + klass.getName());  
271 }  
272 return ctor.newInstance(args);  
273 }  
274  
275 /\*\*  
276 \* Returns a constructor with single argument.  
277 \* @param <T> the type of the constructor  
278 \* @param klass the class to be constructed  
279 \* @param parameterType The constructor parameter type  
280 \* @return null if matching accessible constructor can not be found.  
281 \* @see Class#getConstructor  
282 \* @see #getAccessibleConstructor(java.lang.reflect.Constructor)  
283 \*/  
284 public static <T> Constructor<T> getAccessibleConstructor(  
285 final Class<T> klass,  
286 final Class<?> parameterType) {  
287  
288 final Class<?>[] parameterTypes = { parameterType };  
289 return getAccessibleConstructor(klass, parameterTypes);  
290 }  
291  
292 /\*\*  
293 \* Returns a constructor given a class and signature.  
294 \* @param <T> the type to be constructed  
295 \* @param klass the class to be constructed  
296 \* @param parameterTypes the parameter array  
297 \* @return null if matching accessible constructor can not be found  
298 \* @see Class#getConstructor  
299 \* @see #getAccessibleConstructor(java.lang.reflect.Constructor)  
300 \*/  
301 public static <T> Constructor<T> getAccessibleConstructor(  
302 final Class<T> klass,  
303 final Class<?>[] parameterTypes) {  
304  
305 try {  
306 return getAccessibleConstructor(  
307 klass.getConstructor(parameterTypes));  
308 } catch (final NoSuchMethodException e) {  
309 return (null);  
310 }  
311 }  
312  
313 /\*\*  
314 \* Returns accessible version of the given constructor.  
315 \* @param <T> the type of the constructor  
316 \* @param ctor prototype constructor object.  
317 \* @return <code>null</code> if accessible constructor can not be found.  
318 \* @see java.lang.SecurityManager  
319 \*/  
320 public static <T> Constructor<T> getAccessibleConstructor(final Constructor<T> ctor) {  
321  
322 // Make sure we have a method to check  
323 if (ctor == null) {  
324 return (null);  
325 }  
326  
327 // If the requested method is not public we cannot call it  
328 if (!Modifier.isPublic(ctor.getModifiers())) {  
329 return (null);  
330 }  
331  
332 // If the declaring class is public, we are done  
333 final Class<T> clazz = ctor.getDeclaringClass();  
334 if (Modifier.isPublic(clazz.getModifiers())) {  
335 return (ctor);  
336 }  
337  
338 // what else can we do?  
339 return null;  
340 }  
341  
342 private static Object[] toArray(final Object arg) {  
343 Object[] args = null;  
344 if (arg != null) {  
345 args = new Object[] { arg };  
346 }  
347 return args;  
348 }  
349  
350 // -------------------------------------------------------- Private Methods  
351 /\*\*  
352 \* <p>Find an accessible constructor with compatible parameters.  
353 \* Compatible parameters mean that every method parameter is assignable from  
354 \* the given parameters. In other words, it finds constructor that will take  
355 \* the parameters given.</p>  
356 \*  
357 \* <p>First it checks if there is constructor matching the exact signature.  
358 \* If no such, all the constructors of the class are tested if their signatures  
359 \* are assignment compatible with the parameter types.  
360 \* The first matching constructor is returned.</p>  
361 \*  
362 \* @param <T> the type of the class to be inspected  
363 \* @param clazz find constructor for this class  
364 \* @param parameterTypes find method with compatible parameters  
365 \* @return a valid Constructor object. If there's no matching constructor, returns <code>null</code>.  
366 \*/  
367 private static <T> Constructor<T> getMatchingAccessibleConstructor(  
368 final Class<T> clazz,  
369 final Class<?>[] parameterTypes) {  
370 // see if we can find the method directly  
371 // most of the time this works and it's much faster  
372 try {  
373 final Constructor<T> ctor = clazz.getConstructor(parameterTypes);  
374 try {  
375 //  
376 // XXX Default access superclass workaround  
377 //  
378 // When a public class has a default access superclass  
379 // with public methods, these methods are accessible.  
380 // Calling them from compiled code works fine.  
381 //  
382 // Unfortunately, using reflection to invoke these methods  
383 // seems to (wrongly) to prevent access even when the method  
384 // modifer is public.  
385 //  
386 // The following workaround solves the problem but will only  
387 // work from sufficiently privilages code.  
388 //  
389 // Better workarounds would be greatfully accepted.  
390 //  
391 ctor.setAccessible(true);  
392 } catch (final SecurityException se) {  
393 /\* SWALLOW, if workaround fails don't fret. \*/  
394 }  
395 return ctor;  
396  
397 } catch (final NoSuchMethodException e) { /\* SWALLOW \*/  
398 }  
399  
400 // search through all methods  
401 final int paramSize = parameterTypes.length;  
402 final Constructor<?>[] ctors = clazz.getConstructors();  
403 for (Constructor<?> ctor2 : ctors) {  
404 // compare parameters  
405 final Class<?>[] ctorParams = ctor2.getParameterTypes();  
406 final int ctorParamSize = ctorParams.length;  
407 if (ctorParamSize == paramSize) {  
408 boolean match = true;  
409 for (int n = 0; n < ctorParamSize; n++) {  
410 if (!MethodUtils  
411 .isAssignmentCompatible(  
412 ctorParams[n],  
413 parameterTypes[n])) {  
414 match = false;  
415 break;  
416 }  
417 }  
418  
419 if (match) {  
420 // get accessible version of method  
421 final Constructor<?> ctor = getAccessibleConstructor(ctor2);  
422 if (ctor != null) {  
423 try {  
424 ctor.setAccessible(true);  
425 } catch (final SecurityException se) {  
426 /\* Swallow SecurityException  
427 \* TODO: Why?  
428 \*/  
429 }  
430 @SuppressWarnings("unchecked")  
431 final  
432 // Class.getConstructors() actually returns constructors  
433 // of type T, so it is safe to cast.  
434 Constructor<T> typedCtor = (Constructor<T>) ctor;  
435 return typedCtor;  
436 }  
437 }  
438 }  
439 }  
440  
441 return null;  
442 }  
443  
444}