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 \*/  
017package org.apache.commons.collections4.functors;  
018  
019import java.io.ByteArrayInputStream;  
020import java.io.ByteArrayOutputStream;  
021import java.io.IOException;  
022import java.io.ObjectInputStream;  
023import java.io.ObjectOutputStream;  
024import java.io.Serializable;  
025import java.lang.reflect.InvocationTargetException;  
026import java.lang.reflect.Method;  
027  
028import org.apache.commons.collections4.Factory;  
029import org.apache.commons.collections4.FunctorException;  
030  
031/\*\*  
032 \* Factory implementation that creates a new instance each time based on a prototype.  
033 \* <p>  
034 \* <b>WARNING:</b> from v4.1 onwards {@link Factory} instances returned by  
035 \* {@link #prototypeFactory(Object)} will <b>not</b> be serializable anymore in order  
036 \* to prevent potential remote code execution exploits. Please refer to  
037 \* <a href="https://issues.apache.org/jira/browse/COLLECTIONS-580">COLLECTIONS-580</a>  
038 \* for more details.  
039 \* </p>  
040 \*  
041 \* @since 3.0  
042 \*/  
043public class PrototypeFactory {  
044  
045 /\*\*  
046 \* Factory method that performs validation.  
047 \* <p>  
048 \* Creates a Factory that will return a clone of the same prototype object  
049 \* each time the factory is used. The prototype will be cloned using one of these  
050 \* techniques (in order):  
051 \* </p>  
052 \*  
053 \* <ul>  
054 \* <li>public clone method</li>  
055 \* <li>public copy constructor</li>  
056 \* <li>serialization clone</li>  
057 \* </ul>  
058 \*  
059 \* @param <T> the type the factory creates  
060 \* @param prototype the object to clone each time in the factory  
061 \* @return the <code>prototype</code> factory, or a {@link ConstantFactory#NULL\_INSTANCE} if  
062 \* the {@code prototype} is {@code null}  
063 \* @throws IllegalArgumentException if the prototype cannot be cloned  
064 \*/  
065 @SuppressWarnings("unchecked")  
066 public static <T> Factory<T> prototypeFactory(final T prototype) {  
067 if (prototype == null) {  
068 return ConstantFactory.<T>constantFactory(null);  
069 }  
070 try {  
071 final Method method = prototype.getClass().getMethod("clone", (Class[]) null);  
072 return new PrototypeCloneFactory<>(prototype, method);  
073  
074 } catch (final NoSuchMethodException ex) {  
075 try {  
076 prototype.getClass().getConstructor(new Class<?>[] { prototype.getClass() });  
077 return new InstantiateFactory<>(  
078 (Class<T>) prototype.getClass(),  
079 new Class<?>[] { prototype.getClass() },  
080 new Object[] { prototype });  
081 } catch (final NoSuchMethodException ex2) {  
082 if (prototype instanceof Serializable) {  
083 return (Factory<T>) new PrototypeSerializationFactory<>((Serializable) prototype);  
084 }  
085 }  
086 }  
087 throw new IllegalArgumentException("The prototype must be cloneable via a public clone method");  
088 }  
089  
090 /\*\*  
091 \* Restricted constructor.  
092 \*/  
093 private PrototypeFactory() {  
094 super();  
095 }  
096  
097 // PrototypeCloneFactory  
098 //-----------------------------------------------------------------------  
099 /\*\*  
100 \* PrototypeCloneFactory creates objects by copying a prototype using the clone method.  
101 \*/  
102 static class PrototypeCloneFactory<T> implements Factory<T> {  
103  
104 /\*\* The object to clone each time \*/  
105 private final T iPrototype;  
106 /\*\* The method used to clone \*/  
107 private transient Method iCloneMethod;  
108  
109 /\*\*  
110 \* Constructor to store prototype.  
111 \*/  
112 private PrototypeCloneFactory(final T prototype, final Method method) {  
113 super();  
114 iPrototype = prototype;  
115 iCloneMethod = method;  
116 }  
117  
118 /\*\*  
119 \* Find the Clone method for the class specified.  
120 \*/  
121 private void findCloneMethod() {  
122 try {  
123 iCloneMethod = iPrototype.getClass().getMethod("clone", (Class[]) null);  
124 } catch (final NoSuchMethodException ex) {  
125 throw new IllegalArgumentException("PrototypeCloneFactory: The clone method must exist and be public ");  
126 }  
127 }  
128  
129 /\*\*  
130 \* Creates an object by calling the clone method.  
131 \*  
132 \* @return the new object  
133 \*/  
134 @Override  
135 @SuppressWarnings("unchecked")  
136 public T create() {  
137 // needed for post-serialization  
138 if (iCloneMethod == null) {  
139 findCloneMethod();  
140 }  
141  
142 try {  
143 return (T) iCloneMethod.invoke(iPrototype, (Object[]) null);  
144 } catch (final IllegalAccessException ex) {  
145 throw new FunctorException("PrototypeCloneFactory: Clone method must be public", ex);  
146 } catch (final InvocationTargetException ex) {  
147 throw new FunctorException("PrototypeCloneFactory: Clone method threw an exception", ex);  
148 }  
149 }  
150 }  
151  
152 // PrototypeSerializationFactory  
153 //-----------------------------------------------------------------------  
154 /\*\*  
155 \* PrototypeSerializationFactory creates objects by cloning a prototype using serialization.  
156 \*/  
157 static class PrototypeSerializationFactory<T extends Serializable> implements Factory<T> {  
158  
159 /\*\* The object to clone via serialization each time \*/  
160 private final T iPrototype;  
161  
162 /\*\*  
163 \* Constructor to store prototype  
164 \*/  
165 private PrototypeSerializationFactory(final T prototype) {  
166 super();  
167 iPrototype = prototype;  
168 }  
169  
170 /\*\*  
171 \* Creates an object using serialization.  
172 \*  
173 \* @return the new object  
174 \*/  
175 @Override  
176 @SuppressWarnings("unchecked")  
177 public T create() {  
178 final ByteArrayOutputStream baos = new ByteArrayOutputStream(512);  
179 ByteArrayInputStream bais = null;  
180 try {  
181 final ObjectOutputStream out = new ObjectOutputStream(baos);  
182 out.writeObject(iPrototype);  
183  
184 bais = new ByteArrayInputStream(baos.toByteArray());  
185 final ObjectInputStream in = new ObjectInputStream(bais);  
186 return (T) in.readObject();  
187  
188 } catch (final ClassNotFoundException ex) {  
189 throw new FunctorException(ex);  
190 } catch (final IOException ex) {  
191 throw new FunctorException(ex);  
192 } finally {  
193 try {  
194 if (bais != null) {  
195 bais.close();  
196 }  
197 } catch (final IOException ex) { //NOPMD  
198 // ignore  
199 }  
200 try {  
201 baos.close();  
202 } catch (final IOException ex) { //NOPMD  
203 // ignore  
204 }  
205 }  
206 }  
207 }  
208  
209}