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.util.Map;  
021import java.util.WeakHashMap;  
022  
023/\*\*  
024 \* An instance of this class represents a value that is provided per (thread)  
025 \* context classloader.  
026 \*  
027 \* <p>Occasionally it is necessary to store data in "global" variables  
028 \* (including uses of the Singleton pattern). In applications which have only  
029 \* a single classloader such data can simply be stored as "static" members on  
030 \* some class. When multiple classloaders are involved, however, this approach  
031 \* can fail; in particular, this doesn't work when the code may be run within a  
032 \* servlet container or a j2ee container, and the class on which the static  
033 \* member is defined is loaded via a "shared" classloader that is visible to all  
034 \* components running within the container. This class provides a mechanism for  
035 \* associating data with a ClassLoader instance, which ensures that when the  
036 \* code runs in such a container each component gets its own copy of the  
037 \* "global" variable rather than unexpectedly sharing a single copy of the  
038 \* variable with other components that happen to be running in the same  
039 \* container at the same time (eg servlets or EJBs.)</p>  
040 \*  
041 \* <p>This class is strongly patterned after the java.lang.ThreadLocal  
042 \* class, which performs a similar task in allowing data to be associated  
043 \* with a particular thread.</p>  
044 \*  
045 \* <p>When code that uses this class is run as a "normal" application, ie  
046 \* not within a container, the effect is identical to just using a static  
047 \* member variable to store the data, because Thread.getContextClassLoader  
048 \* always returns the same classloader (the system classloader).</p>  
049 \*  
050 \* <p>Expected usage is as follows:<br>  
051 \* <pre>  
052 \* public class SomeClass {  
053 \* private static final ContextClassLoaderLocal<String> global  
054 \* = new ContextClassLoaderLocal<String>() {  
055 \* protected String initialValue() {  
056 \* return new String("Initial value");  
057 \* };  
058 \*  
059 \* public void testGlobal() {  
060 \* String s = global.get();  
061 \* System.out.println("global value:" + s);  
062 \* buf.set("New Value");  
063 \* }  
064 \* </pre>  
065 \* </p>  
066 \*  
067 \* <p><strong>Note:</strong> This class takes some care to ensure that when  
068 \* a component which uses this class is "undeployed" by a container the  
069 \* component-specific classloader and all its associated classes (and their  
070 \* static variables) are garbage-collected. Unfortunately there is one  
071 \* scenario in which this does <i>not</i> work correctly and there  
072 \* is unfortunately no known workaround other than ensuring that the  
073 \* component (or its container) calls the "unset" method on this class for  
074 \* each instance of this class when the component is undeployed. The problem  
075 \* occurs if:  
076 \* <ul>  
077 \* <li>the class containing a static instance of this class was loaded via  
078 \* a shared classloader, and</li>  
079 \* <li>the value stored in the instance is an object whose class was loaded  
080 \* via the component-specific classloader (or any of the objects it refers  
081 \* to were loaded via that classloader).</li>  
082 \* </ul>  
083 \* The result is that the map managed by this object still contains a strong  
084 \* reference to the stored object, which contains a strong reference to the  
085 \* classloader that loaded it, meaning that although the container has  
086 \* "undeployed" the component the component-specific classloader and all the  
087 \* related classes and static variables cannot be garbage-collected. This is  
088 \* not expected to be an issue with the commons-beanutils library as the only  
089 \* classes which use this class are BeanUtilsBean and ConvertUtilsBean and  
090 \* there is no obvious reason for a user of the beanutils library to subclass  
091 \* either of those classes.</p>  
092 \*  
093 \* <p><strong>Note:</strong> A WeakHashMap bug in several 1.3 JVMs results in  
094 \* a memory leak for those JVMs.</p>  
095 \*  
096 \* <p><strong>Note:</strong> Of course all of this would be unnecessary if  
097 \* containers required each component to load the full set of classes it  
098 \* needs, ie avoided providing classes loaded via a "shared" classloader.</p>  
099 \*  
100 \* @param <T> the type of data stored in an instance  
101 \* @version $Id$  
102 \* @see java.lang.Thread#getContextClassLoader  
103 \*/  
104public class ContextClassLoaderLocal<T> {  
105 private final Map<ClassLoader, T> valueByClassLoader = new WeakHashMap<ClassLoader, T>();  
106 private boolean globalValueInitialized = false;  
107 private T globalValue;  
108  
109 /\*\*  
110 \* Construct a context classloader instance  
111 \*/  
112 public ContextClassLoaderLocal() {  
113 super();  
114 }  
115  
116 /\*\*  
117 \* Returns the initial value for this ContextClassLoaderLocal  
118 \* variable. This method will be called once per Context ClassLoader for  
119 \* each ContextClassLoaderLocal, the first time it is accessed  
120 \* with get or set. If the programmer desires ContextClassLoaderLocal variables  
121 \* to be initialized to some value other than null, ContextClassLoaderLocal must  
122 \* be subclassed, and this method overridden. Typically, an anonymous  
123 \* inner class will be used. Typical implementations of initialValue  
124 \* will call an appropriate constructor and return the newly constructed  
125 \* object.  
126 \*  
127 \* @return a new Object to be used as an initial value for this ContextClassLoaderLocal  
128 \*/  
129 protected T initialValue() {  
130 return null;  
131 }  
132  
133 /\*\*  
134 \* Gets the instance which provides the functionality for {@link BeanUtils}.  
135 \* This is a pseudo-singleton - an single instance is provided per (thread) context classloader.  
136 \* This mechanism provides isolation for web apps deployed in the same container.  
137 \* @return the object currently associated with the context-classloader of the current thread.  
138 \*/  
139 public synchronized T get() {  
140 // synchronizing the whole method is a bit slower  
141 // but guarantees no subtle threading problems, and there's no  
142 // need to synchronize valueByClassLoader  
143  
144 // make sure that the map is given a change to purge itself  
145 valueByClassLoader.isEmpty();  
146 try {  
147  
148 final ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();  
149 if (contextClassLoader != null) {  
150  
151 T value = valueByClassLoader.get(contextClassLoader);  
152 if ((value == null)  
153 && !valueByClassLoader.containsKey(contextClassLoader)) {  
154 value = initialValue();  
155 valueByClassLoader.put(contextClassLoader, value);  
156 }  
157 return value;  
158  
159 }  
160  
161 } catch (final SecurityException e) { /\* SWALLOW - should we log this? \*/ }  
162  
163 // if none or exception, return the globalValue  
164 if (!globalValueInitialized) {  
165 globalValue = initialValue();  
166 globalValueInitialized = true;  
167 }//else already set  
168 return globalValue;  
169 }  
170  
171 /\*\*  
172 \* Sets the value - a value is provided per (thread) context classloader.  
173 \* This mechanism provides isolation for web apps deployed in the same container.  
174 \*  
175 \* @param value the object to be associated with the entrant thread's context classloader  
176 \*/  
177 public synchronized void set(final T value) {  
178 // synchronizing the whole method is a bit slower  
179 // but guarentees no subtle threading problems  
180  
181 // make sure that the map is given a change to purge itself  
182 valueByClassLoader.isEmpty();  
183 try {  
184  
185 final ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();  
186 if (contextClassLoader != null) {  
187 valueByClassLoader.put(contextClassLoader, value);  
188 return;  
189 }  
190  
191 } catch (final SecurityException e) { /\* SWALLOW - should we log this? \*/ }  
192  
193 // if in doubt, set the global value  
194 globalValue = value;  
195 globalValueInitialized = true;  
196 }  
197  
198 /\*\*  
199 \* Unsets the value associated with the current thread's context classloader  
200 \*/  
201 public synchronized void unset() {  
202 try {  
203  
204 final ClassLoader contextClassLoader = Thread.currentThread().getContextClassLoader();  
205 unset(contextClassLoader);  
206  
207 } catch (final SecurityException e) { /\* SWALLOW - should we log this? \*/ }  
208 }  
209  
210 /\*\*  
211 \* Unsets the value associated with the given classloader  
212 \* @param classLoader The classloader to <i>unset</i> for  
213 \*/  
214 public synchronized void unset(final ClassLoader classLoader) {  
215 valueByClassLoader.remove(classLoader);  
216 }  
217}