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.map;  
018  
019import java.io.Serializable;  
020import java.util.AbstractSet;  
021import java.util.Collection;  
022import java.util.Collections;  
023import java.util.Iterator;  
024import java.util.Map;  
025import java.util.NoSuchElementException;  
026import java.util.Set;  
027  
028import org.apache.commons.collections4.BoundedMap;  
029import org.apache.commons.collections4.KeyValue;  
030import org.apache.commons.collections4.OrderedMap;  
031import org.apache.commons.collections4.OrderedMapIterator;  
032import org.apache.commons.collections4.ResettableIterator;  
033import org.apache.commons.collections4.iterators.SingletonIterator;  
034import org.apache.commons.collections4.keyvalue.TiedMapEntry;  
035  
036/\*\*  
037 \* A <code>Map</code> implementation that holds a single item and is fixed size.  
038 \* <p>  
039 \* The single key/value pair is specified at creation.  
040 \* The map is fixed size so any action that would change the size is disallowed.  
041 \* However, the <code>put</code> or <code>setValue</code> methods can <i>change</i>  
042 \* the value associated with the key.  
043 \* </p>  
044 \* <p>  
045 \* If trying to remove or clear the map, an UnsupportedOperationException is thrown.  
046 \* If trying to put a new mapping into the map, an IllegalArgumentException is thrown.  
047 \* The put method will only succeed if the key specified is the same as the  
048 \* singleton key.  
049 \* </p>  
050 \* <p>  
051 \* The key and value can be obtained by:  
052 \* </p>  
053 \* <ul>  
054 \* <li>normal Map methods and views  
055 \* <li>the <code>MapIterator</code>, see {@link #mapIterator()}  
056 \* <li>the <code>KeyValue</code> interface (just cast - no object creation)  
057 \* </ul>  
058 \*  
059 \* @param <K> the type of the keys in this map  
060 \* @param <V> the type of the values in this map  
061 \* @since 3.1  
062 \*/  
063public class SingletonMap<K, V>  
064 implements OrderedMap<K, V>, BoundedMap<K, V>, KeyValue<K, V>, Serializable, Cloneable {  
065  
066 /\*\* Serialization version \*/  
067 private static final long serialVersionUID = -8931271118676803261L;  
068  
069 /\*\* Singleton key \*/  
070 private final K key;  
071 /\*\* Singleton value \*/  
072 private V value;  
073  
074 /\*\*  
075 \* Constructor that creates a map of <code>null</code> to <code>null</code>.  
076 \*/  
077 public SingletonMap() {  
078 super();  
079 this.key = null;  
080 }  
081  
082 /\*\*  
083 \* Constructor specifying the key and value.  
084 \*  
085 \* @param key the key to use  
086 \* @param value the value to use  
087 \*/  
088 public SingletonMap(final K key, final V value) {  
089 super();  
090 this.key = key;  
091 this.value = value;  
092 }  
093  
094 /\*\*  
095 \* Constructor specifying the key and value as a <code>KeyValue</code>.  
096 \*  
097 \* @param keyValue the key value pair to use  
098 \*/  
099 public SingletonMap(final KeyValue<K, V> keyValue) {  
100 super();  
101 this.key = keyValue.getKey();  
102 this.value = keyValue.getValue();  
103 }  
104  
105 /\*\*  
106 \* Constructor specifying the key and value as a <code>MapEntry</code>.  
107 \*  
108 \* @param mapEntry the mapEntry to use  
109 \*/  
110 public SingletonMap(final Map.Entry<? extends K, ? extends V> mapEntry) {  
111 super();  
112 this.key = mapEntry.getKey();  
113 this.value = mapEntry.getValue();  
114 }  
115  
116 /\*\*  
117 \* Constructor copying elements from another map.  
118 \*  
119 \* @param map the map to copy, must be size 1  
120 \* @throws NullPointerException if the map is null  
121 \* @throws IllegalArgumentException if the size is not 1  
122 \*/  
123 public SingletonMap(final Map<? extends K, ? extends V> map) {  
124 super();  
125 if (map.size() != 1) {  
126 throw new IllegalArgumentException("The map size must be 1");  
127 }  
128 final Map.Entry<? extends K, ? extends V> entry = map.entrySet().iterator().next();  
129 this.key = entry.getKey();  
130 this.value = entry.getValue();  
131 }  
132  
133 // KeyValue  
134 //-----------------------------------------------------------------------  
135 /\*\*  
136 \* Gets the key.  
137 \*  
138 \* @return the key  
139 \*/  
140 @Override  
141 public K getKey() {  
142 return key;  
143 }  
144  
145 /\*\*  
146 \* Gets the value.  
147 \*  
148 \* @return the value  
149 \*/  
150 @Override  
151 public V getValue() {  
152 return value;  
153 }  
154  
155 /\*\*  
156 \* Sets the value.  
157 \*  
158 \* @param value the new value to set  
159 \* @return the old value  
160 \*/  
161 public V setValue(final V value) {  
162 final V old = this.value;  
163 this.value = value;  
164 return old;  
165 }  
166  
167 // BoundedMap  
168 //-----------------------------------------------------------------------  
169 /\*\*  
170 \* Is the map currently full, always true.  
171 \*  
172 \* @return true always  
173 \*/  
174 @Override  
175 public boolean isFull() {  
176 return true;  
177 }  
178  
179 /\*\*  
180 \* Gets the maximum size of the map, always 1.  
181 \*  
182 \* @return 1 always  
183 \*/  
184 @Override  
185 public int maxSize() {  
186 return 1;  
187 }  
188  
189 // Map  
190 //-----------------------------------------------------------------------  
191 /\*\*  
192 \* Gets the value mapped to the key specified.  
193 \*  
194 \* @param key the key  
195 \* @return the mapped value, null if no match  
196 \*/  
197 @Override  
198 public V get(final Object key) {  
199 if (isEqualKey(key)) {  
200 return value;  
201 }  
202 return null;  
203 }  
204  
205 /\*\*  
206 \* Gets the size of the map, always 1.  
207 \*  
208 \* @return the size of 1  
209 \*/  
210 @Override  
211 public int size() {  
212 return 1;  
213 }  
214  
215 /\*\*  
216 \* Checks whether the map is currently empty, which it never is.  
217 \*  
218 \* @return false always  
219 \*/  
220 @Override  
221 public boolean isEmpty() {  
222 return false;  
223 }  
224  
225 //-----------------------------------------------------------------------  
226 /\*\*  
227 \* Checks whether the map contains the specified key.  
228 \*  
229 \* @param key the key to search for  
230 \* @return true if the map contains the key  
231 \*/  
232 @Override  
233 public boolean containsKey(final Object key) {  
234 return isEqualKey(key);  
235 }  
236  
237 /\*\*  
238 \* Checks whether the map contains the specified value.  
239 \*  
240 \* @param value the value to search for  
241 \* @return true if the map contains the key  
242 \*/  
243 @Override  
244 public boolean containsValue(final Object value) {  
245 return isEqualValue(value);  
246 }  
247  
248 //-----------------------------------------------------------------------  
249 /\*\*  
250 \* Puts a key-value mapping into this map where the key must match the existing key.  
251 \* <p>  
252 \* An IllegalArgumentException is thrown if the key does not match as the map  
253 \* is fixed size.  
254 \*  
255 \* @param key the key to set, must be the key of the map  
256 \* @param value the value to set  
257 \* @return the value previously mapped to this key, null if none  
258 \* @throws IllegalArgumentException if the key does not match  
259 \*/  
260 @Override  
261 public V put(final K key, final V value) {  
262 if (isEqualKey(key)) {  
263 return setValue(value);  
264 }  
265 throw new IllegalArgumentException("Cannot put new key/value pair - Map is fixed size singleton");  
266 }  
267  
268 /\*\*  
269 \* Puts the values from the specified map into this map.  
270 \* <p>  
271 \* The map must be of size 0 or size 1.  
272 \* If it is size 1, the key must match the key of this map otherwise an  
273 \* IllegalArgumentException is thrown.  
274 \*  
275 \* @param map the map to add, must be size 0 or 1, and the key must match  
276 \* @throws NullPointerException if the map is null  
277 \* @throws IllegalArgumentException if the key does not match  
278 \*/  
279 @Override  
280 public void putAll(final Map<? extends K, ? extends V> map) {  
281 switch (map.size()) {  
282 case 0:  
283 return;  
284  
285 case 1:  
286 final Map.Entry<? extends K, ? extends V> entry = map.entrySet().iterator().next();  
287 put(entry.getKey(), entry.getValue());  
288 return;  
289  
290 default:  
291 throw new IllegalArgumentException("The map size must be 0 or 1");  
292 }  
293 }  
294  
295 /\*\*  
296 \* Unsupported operation.  
297 \*  
298 \* @param key the mapping to remove  
299 \* @return the value mapped to the removed key, null if key not in map  
300 \* @throws UnsupportedOperationException always  
301 \*/  
302 @Override  
303 public V remove(final Object key) {  
304 throw new UnsupportedOperationException();  
305 }  
306  
307 /\*\*  
308 \* Unsupported operation.  
309 \*/  
310 @Override  
311 public void clear() {  
312 throw new UnsupportedOperationException();  
313 }  
314  
315 //-----------------------------------------------------------------------  
316 /\*\*  
317 \* Gets the entrySet view of the map.  
318 \* Changes made via <code>setValue</code> affect this map.  
319 \* To simply iterate through the entries, use {@link #mapIterator()}.  
320 \*  
321 \* @return the entrySet view  
322 \*/  
323 @Override  
324 public Set<Map.Entry<K, V>> entrySet() {  
325 final Map.Entry<K, V> entry = new TiedMapEntry<>(this, getKey());  
326 return Collections.singleton(entry);  
327 }  
328  
329 /\*\*  
330 \* Gets the unmodifiable keySet view of the map.  
331 \* Changes made to the view affect this map.  
332 \* To simply iterate through the keys, use {@link #mapIterator()}.  
333 \*  
334 \* @return the keySet view  
335 \*/  
336 @Override  
337 public Set<K> keySet() {  
338 return Collections.singleton(key);  
339 }  
340  
341 /\*\*  
342 \* Gets the unmodifiable values view of the map.  
343 \* Changes made to the view affect this map.  
344 \* To simply iterate through the values, use {@link #mapIterator()}.  
345 \*  
346 \* @return the values view  
347 \*/  
348 @Override  
349 public Collection<V> values() {  
350 return new SingletonValues<>(this);  
351 }  
352  
353 /\*\*  
354 \* {@inheritDoc}  
355 \*/  
356 @Override  
357 public OrderedMapIterator<K, V> mapIterator() {  
358 return new SingletonMapIterator<>(this);  
359 }  
360  
361 /\*\*  
362 \* Gets the first (and only) key in the map.  
363 \*  
364 \* @return the key  
365 \*/  
366 @Override  
367 public K firstKey() {  
368 return getKey();  
369 }  
370  
371 /\*\*  
372 \* Gets the last (and only) key in the map.  
373 \*  
374 \* @return the key  
375 \*/  
376 @Override  
377 public K lastKey() {  
378 return getKey();  
379 }  
380  
381 /\*\*  
382 \* Gets the next key after the key specified, always null.  
383 \*  
384 \* @param key the next key  
385 \* @return null always  
386 \*/  
387 @Override  
388 public K nextKey(final K key) {  
389 return null;  
390 }  
391  
392 /\*\*  
393 \* Gets the previous key before the key specified, always null.  
394 \*  
395 \* @param key the next key  
396 \* @return null always  
397 \*/  
398 @Override  
399 public K previousKey(final K key) {  
400 return null;  
401 }  
402  
403 //-----------------------------------------------------------------------  
404 /\*\*  
405 \* Compares the specified key to the stored key.  
406 \*  
407 \* @param key the key to compare  
408 \* @return true if equal  
409 \*/  
410 protected boolean isEqualKey(final Object key) {  
411 return key == null ? getKey() == null : key.equals(getKey());  
412 }  
413  
414 /\*\*  
415 \* Compares the specified value to the stored value.  
416 \*  
417 \* @param value the value to compare  
418 \* @return true if equal  
419 \*/  
420 protected boolean isEqualValue(final Object value) {  
421 return value == null ? getValue() == null : value.equals(getValue());  
422 }  
423  
424 //-----------------------------------------------------------------------  
425 /\*\*  
426 \* SingletonMapIterator.  
427 \*/  
428 static class SingletonMapIterator<K, V> implements OrderedMapIterator<K, V>, ResettableIterator<K> {  
429 private final SingletonMap<K, V> parent;  
430 private boolean hasNext = true;  
431 private boolean canGetSet = false;  
432  
433 SingletonMapIterator(final SingletonMap<K, V> parent) {  
434 super();  
435 this.parent = parent;  
436 }  
437  
438 @Override  
439 public boolean hasNext() {  
440 return hasNext;  
441 }  
442  
443 @Override  
444 public K next() {  
445 if (hasNext == false) {  
446 throw new NoSuchElementException(AbstractHashedMap.NO\_NEXT\_ENTRY);  
447 }  
448 hasNext = false;  
449 canGetSet = true;  
450 return parent.getKey();  
451 }  
452  
453 @Override  
454 public boolean hasPrevious() {  
455 return hasNext == false;  
456 }  
457  
458 @Override  
459 public K previous() {  
460 if (hasNext == true) {  
461 throw new NoSuchElementException(AbstractHashedMap.NO\_PREVIOUS\_ENTRY);  
462 }  
463 hasNext = true;  
464 return parent.getKey();  
465 }  
466  
467 @Override  
468 public void remove() {  
469 throw new UnsupportedOperationException();  
470 }  
471  
472 @Override  
473 public K getKey() {  
474 if (canGetSet == false) {  
475 throw new IllegalStateException(AbstractHashedMap.GETKEY\_INVALID);  
476 }  
477 return parent.getKey();  
478 }  
479  
480 @Override  
481 public V getValue() {  
482 if (canGetSet == false) {  
483 throw new IllegalStateException(AbstractHashedMap.GETVALUE\_INVALID);  
484 }  
485 return parent.getValue();  
486 }  
487  
488 @Override  
489 public V setValue(final V value) {  
490 if (canGetSet == false) {  
491 throw new IllegalStateException(AbstractHashedMap.SETVALUE\_INVALID);  
492 }  
493 return parent.setValue(value);  
494 }  
495  
496 @Override  
497 public void reset() {  
498 hasNext = true;  
499 }  
500  
501 @Override  
502 public String toString() {  
503 if (hasNext) {  
504 return "Iterator[]";  
505 }  
506 return "Iterator[" + getKey() + "=" + getValue() + "]";  
507 }  
508 }  
509  
510 /\*\*  
511 \* Values implementation for the SingletonMap.  
512 \* This class is needed as values is a view that must update as the map updates.  
513 \*/  
514 static class SingletonValues<V> extends AbstractSet<V> implements Serializable {  
515 private static final long serialVersionUID = -3689524741863047872L;  
516 private final SingletonMap<?, V> parent;  
517  
518 SingletonValues(final SingletonMap<?, V> parent) {  
519 super();  
520 this.parent = parent;  
521 }  
522  
523 @Override  
524 public int size() {  
525 return 1;  
526 }  
527 @Override  
528 public boolean isEmpty() {  
529 return false;  
530 }  
531 @Override  
532 public boolean contains(final Object object) {  
533 return parent.containsValue(object);  
534 }  
535 @Override  
536 public void clear() {  
537 throw new UnsupportedOperationException();  
538 }  
539 @Override  
540 public Iterator<V> iterator() {  
541 return new SingletonIterator<>(parent.getValue(), false);  
542 }  
543 }  
544  
545 //-----------------------------------------------------------------------  
546 /\*\*  
547 \* Clones the map without cloning the key or value.  
548 \*  
549 \* @return a shallow clone  
550 \*/  
551 @Override  
552 @SuppressWarnings("unchecked")  
553 public SingletonMap<K, V> clone() {  
554 try {  
555 return (SingletonMap<K, V>) super.clone();  
556 } catch (final CloneNotSupportedException ex) {  
557 throw new InternalError();  
558 }  
559 }  
560  
561 /\*\*  
562 \* Compares this map with another.  
563 \*  
564 \* @param obj the object to compare to  
565 \* @return true if equal  
566 \*/  
567 @Override  
568 public boolean equals(final Object obj) {  
569 if (obj == this) {  
570 return true;  
571 }  
572 if (obj instanceof Map == false) {  
573 return false;  
574 }  
575 final Map<?,?> other = (Map<?,?>) obj;  
576 if (other.size() != 1) {  
577 return false;  
578 }  
579 final Map.Entry<?,?> entry = other.entrySet().iterator().next();  
580 return isEqualKey(entry.getKey()) && isEqualValue(entry.getValue());  
581 }  
582  
583 /\*\*  
584 \* Gets the standard Map hashCode.  
585 \*  
586 \* @return the hash code defined in the Map interface  
587 \*/  
588 @Override  
589 public int hashCode() {  
590 return (getKey() == null ? 0 : getKey().hashCode()) ^  
591 (getValue() == null ? 0 : getValue().hashCode());  
592 }  
593  
594 /\*\*  
595 \* Gets the map as a String.  
596 \*  
597 \* @return a string version of the map  
598 \*/  
599 @Override  
600 public String toString() {  
601 return new StringBuilder(128)  
602 .append('{')  
603 .append(getKey() == this ? "(this Map)" : getKey())  
604 .append('=')  
605 .append(getValue() == this ? "(this Map)" : getValue())  
606 .append('}')  
607 .toString();  
608 }  
609  
610}