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;  
018  
019import java.io.PrintStream;  
020import java.text.NumberFormat;  
021import java.text.ParseException;  
022import java.util.ArrayDeque;  
023import java.util.Collection;  
024import java.util.Collections;  
025import java.util.Deque;  
026import java.util.Enumeration;  
027import java.util.HashMap;  
028import java.util.Iterator;  
029import java.util.Map;  
030import java.util.Map.Entry;  
031import java.util.Properties;  
032import java.util.ResourceBundle;  
033import java.util.SortedMap;  
034import java.util.TreeMap;  
035  
036import org.apache.commons.collections4.map.AbstractMapDecorator;  
037import org.apache.commons.collections4.map.AbstractSortedMapDecorator;  
038import org.apache.commons.collections4.map.FixedSizeMap;  
039import org.apache.commons.collections4.map.FixedSizeSortedMap;  
040import org.apache.commons.collections4.map.LazyMap;  
041import org.apache.commons.collections4.map.LazySortedMap;  
042import org.apache.commons.collections4.map.ListOrderedMap;  
043import org.apache.commons.collections4.map.MultiValueMap;  
044import org.apache.commons.collections4.map.PredicatedMap;  
045import org.apache.commons.collections4.map.PredicatedSortedMap;  
046import org.apache.commons.collections4.map.TransformedMap;  
047import org.apache.commons.collections4.map.TransformedSortedMap;  
048import org.apache.commons.collections4.map.UnmodifiableMap;  
049import org.apache.commons.collections4.map.UnmodifiableSortedMap;  
050  
051/\*\*  
052 \* Provides utility methods and decorators for  
053 \* {@link Map} and {@link SortedMap} instances.  
054 \* <p>  
055 \* It contains various type safe methods  
056 \* as well as other useful features like deep copying.  
057 \* </p>  
058 \* <p>  
059 \* It also provides the following decorators:  
060 \* </p>  
061 \*  
062 \* <ul>  
063 \* <li>{@link #fixedSizeMap(Map)}  
064 \* <li>{@link #fixedSizeSortedMap(SortedMap)}  
065 \* <li>{@link #lazyMap(Map,Factory)}  
066 \* <li>{@link #lazyMap(Map,Transformer)}  
067 \* <li>{@link #lazySortedMap(SortedMap,Factory)}  
068 \* <li>{@link #lazySortedMap(SortedMap,Transformer)}  
069 \* <li>{@link #predicatedMap(Map,Predicate,Predicate)}  
070 \* <li>{@link #predicatedSortedMap(SortedMap,Predicate,Predicate)}  
071 \* <li>{@link #transformedMap(Map, Transformer, Transformer)}  
072 \* <li>{@link #transformedSortedMap(SortedMap, Transformer, Transformer)}  
073 \* <li>{@link #multiValueMap( Map )}  
074 \* <li>{@link #multiValueMap( Map, Class )}  
075 \* <li>{@link #multiValueMap( Map, Factory )}  
076 \* </ul>  
077 \*  
078 \* @since 1.0  
079 \*/  
080@SuppressWarnings("deprecation")  
081public class MapUtils {  
082  
083 /\*\*  
084 \* An empty unmodifiable sorted map.  
085 \* This is not provided in the JDK.  
086 \*/  
087 @SuppressWarnings("rawtypes")  
088 public static final SortedMap EMPTY\_SORTED\_MAP =  
089 UnmodifiableSortedMap.unmodifiableSortedMap(new TreeMap<>());  
090  
091 /\*\*  
092 \* String used to indent the verbose and debug Map prints.  
093 \*/  
094 private static final String INDENT\_STRING = " ";  
095  
096 /\*\*  
097 \* <code>MapUtils</code> should not normally be instantiated.  
098 \*/  
099 private MapUtils() {}  
100  
101 // Type safe getters  
102 //-------------------------------------------------------------------------  
103 /\*\*  
104 \* Gets from a Map in a null-safe manner.  
105 \*  
106 \* @param <K> the key type  
107 \* @param <V> the value type  
108 \* @param map the map to use  
109 \* @param key the key to look up  
110 \* @return the value in the Map, <code>null</code> if null map input  
111 \*/  
112 public static <K, V> V getObject(final Map<? super K, V> map, final K key) {  
113 if (map != null) {  
114 return map.get(key);  
115 }  
116 return null;  
117 }  
118  
119 /\*\*  
120 \* Gets a String from a Map in a null-safe manner.  
121 \* <p>  
122 \* The String is obtained via <code>toString</code>.  
123 \*  
124 \* @param <K> the key type  
125 \* @param map the map to use  
126 \* @param key the key to look up  
127 \* @return the value in the Map as a String, <code>null</code> if null map input  
128 \*/  
129 public static <K> String getString(final Map<? super K, ?> map, final K key) {  
130 if (map != null) {  
131 final Object answer = map.get(key);  
132 if (answer != null) {  
133 return answer.toString();  
134 }  
135 }  
136 return null;  
137 }  
138  
139 /\*\*  
140 \* Gets a Boolean from a Map in a null-safe manner.  
141 \* <p>  
142 \* If the value is a <code>Boolean</code> it is returned directly.  
143 \* If the value is a <code>String</code> and it equals 'true' ignoring case  
144 \* then <code>true</code> is returned, otherwise <code>false</code>.  
145 \* If the value is a <code>Number</code> an integer zero value returns  
146 \* <code>false</code> and non-zero returns <code>true</code>.  
147 \* Otherwise, <code>null</code> is returned.  
148 \*  
149 \* @param <K> the key type  
150 \* @param map the map to use  
151 \* @param key the key to look up  
152 \* @return the value in the Map as a Boolean, <code>null</code> if null map input  
153 \*/  
154 public static <K> Boolean getBoolean(final Map<? super K, ?> map, final K key) {  
155 if (map != null) {  
156 final Object answer = map.get(key);  
157 if (answer != null) {  
158 if (answer instanceof Boolean) {  
159 return (Boolean) answer;  
160 }  
161 if (answer instanceof String) {  
162 return Boolean.valueOf((String) answer);  
163 }  
164 if (answer instanceof Number) {  
165 final Number n = (Number) answer;  
166 return n.intValue() != 0 ? Boolean.TRUE : Boolean.FALSE;  
167 }  
168 }  
169 }  
170 return null;  
171 }  
172  
173 /\*\*  
174 \* Gets a Number from a Map in a null-safe manner.  
175 \* <p>  
176 \* If the value is a <code>Number</code> it is returned directly.  
177 \* If the value is a <code>String</code> it is converted using  
178 \* {@link NumberFormat#parse(String)} on the system default formatter  
179 \* returning <code>null</code> if the conversion fails.  
180 \* Otherwise, <code>null</code> is returned.  
181 \*  
182 \* @param <K> the key type  
183 \* @param map the map to use  
184 \* @param key the key to look up  
185 \* @return the value in the Map as a Number, <code>null</code> if null map input  
186 \*/  
187 public static <K> Number getNumber(final Map<? super K, ?> map, final K key) {  
188 if (map != null) {  
189 final Object answer = map.get(key);  
190 if (answer != null) {  
191 if (answer instanceof Number) {  
192 return (Number) answer;  
193 }  
194 if (answer instanceof String) {  
195 try {  
196 final String text = (String) answer;  
197 return NumberFormat.getInstance().parse(text);  
198 } catch (final ParseException e) { // NOPMD  
199 // failure means null is returned  
200 }  
201 }  
202 }  
203 }  
204 return null;  
205 }  
206  
207 /\*\*  
208 \* Gets a Byte from a Map in a null-safe manner.  
209 \* <p>  
210 \* The Byte is obtained from the results of {@link #getNumber(Map,Object)}.  
211 \*  
212 \* @param <K> the key type  
213 \* @param map the map to use  
214 \* @param key the key to look up  
215 \* @return the value in the Map as a Byte, <code>null</code> if null map input  
216 \*/  
217 public static <K> Byte getByte(final Map<? super K, ?> map, final K key) {  
218 final Number answer = getNumber(map, key);  
219 if (answer == null) {  
220 return null;  
221 }  
222 if (answer instanceof Byte) {  
223 return (Byte) answer;  
224 }  
225 return Byte.valueOf(answer.byteValue());  
226 }  
227  
228 /\*\*  
229 \* Gets a Short from a Map in a null-safe manner.  
230 \* <p>  
231 \* The Short is obtained from the results of {@link #getNumber(Map,Object)}.  
232 \*  
233 \* @param <K> the key type  
234 \* @param map the map to use  
235 \* @param key the key to look up  
236 \* @return the value in the Map as a Short, <code>null</code> if null map input  
237 \*/  
238 public static <K> Short getShort(final Map<? super K, ?> map, final K key) {  
239 final Number answer = getNumber(map, key);  
240 if (answer == null) {  
241 return null;  
242 }  
243 if (answer instanceof Short) {  
244 return (Short) answer;  
245 }  
246 return Short.valueOf(answer.shortValue());  
247 }  
248  
249 /\*\*  
250 \* Gets a Integer from a Map in a null-safe manner.  
251 \* <p>  
252 \* The Integer is obtained from the results of {@link #getNumber(Map,Object)}.  
253 \*  
254 \* @param <K> the key type  
255 \* @param map the map to use  
256 \* @param key the key to look up  
257 \* @return the value in the Map as a Integer, <code>null</code> if null map input  
258 \*/  
259 public static <K> Integer getInteger(final Map<? super K, ?> map, final K key) {  
260 final Number answer = getNumber(map, key);  
261 if (answer == null) {  
262 return null;  
263 }  
264 if (answer instanceof Integer) {  
265 return (Integer) answer;  
266 }  
267 return Integer.valueOf(answer.intValue());  
268 }  
269  
270 /\*\*  
271 \* Gets a Long from a Map in a null-safe manner.  
272 \* <p>  
273 \* The Long is obtained from the results of {@link #getNumber(Map,Object)}.  
274 \*  
275 \* @param <K> the key type  
276 \* @param map the map to use  
277 \* @param key the key to look up  
278 \* @return the value in the Map as a Long, <code>null</code> if null map input  
279 \*/  
280 public static <K> Long getLong(final Map<? super K, ?> map, final K key) {  
281 final Number answer = getNumber(map, key);  
282 if (answer == null) {  
283 return null;  
284 }  
285 if (answer instanceof Long) {  
286 return (Long) answer;  
287 }  
288 return Long.valueOf(answer.longValue());  
289 }  
290  
291 /\*\*  
292 \* Gets a Float from a Map in a null-safe manner.  
293 \* <p>  
294 \* The Float is obtained from the results of {@link #getNumber(Map,Object)}.  
295 \*  
296 \* @param <K> the key type  
297 \* @param map the map to use  
298 \* @param key the key to look up  
299 \* @return the value in the Map as a Float, <code>null</code> if null map input  
300 \*/  
301 public static <K> Float getFloat(final Map<? super K, ?> map, final K key) {  
302 final Number answer = getNumber(map, key);  
303 if (answer == null) {  
304 return null;  
305 }  
306 if (answer instanceof Float) {  
307 return (Float) answer;  
308 }  
309 return Float.valueOf(answer.floatValue());  
310 }  
311  
312 /\*\*  
313 \* Gets a Double from a Map in a null-safe manner.  
314 \* <p>  
315 \* The Double is obtained from the results of {@link #getNumber(Map,Object)}.  
316 \*  
317 \* @param <K> the key type  
318 \* @param map the map to use  
319 \* @param key the key to look up  
320 \* @return the value in the Map as a Double, <code>null</code> if null map input  
321 \*/  
322 public static <K> Double getDouble(final Map<? super K, ?> map, final K key) {  
323 final Number answer = getNumber(map, key);  
324 if (answer == null) {  
325 return null;  
326 }  
327 if (answer instanceof Double) {  
328 return (Double) answer;  
329 }  
330 return Double.valueOf(answer.doubleValue());  
331 }  
332  
333 /\*\*  
334 \* Gets a Map from a Map in a null-safe manner.  
335 \* <p>  
336 \* If the value returned from the specified map is not a Map then  
337 \* <code>null</code> is returned.  
338 \*  
339 \* @param <K> the key type  
340 \* @param map the map to use  
341 \* @param key the key to look up  
342 \* @return the value in the Map as a Map, <code>null</code> if null map input  
343 \*/  
344 public static <K> Map<?, ?> getMap(final Map<? super K, ?> map, final K key) {  
345 if (map != null) {  
346 final Object answer = map.get(key);  
347 if (answer != null && answer instanceof Map) {  
348 return (Map<?, ?>) answer;  
349 }  
350 }  
351 return null;  
352 }  
353  
354 // Type safe getters with default values  
355 //-------------------------------------------------------------------------  
356 /\*\*  
357 \* Looks up the given key in the given map, converting null into the  
358 \* given default value.  
359 \*  
360 \* @param <K> the key type  
361 \* @param <V> the value type  
362 \* @param map the map whose value to look up  
363 \* @param key the key of the value to look up in that map  
364 \* @param defaultValue what to return if the value is null  
365 \* @return the value in the map, or defaultValue if the original value  
366 \* is null or the map is null  
367 \*/  
368 public static <K, V> V getObject(final Map<K, V> map, final K key, final V defaultValue) {  
369 if (map != null) {  
370 final V answer = map.get(key);  
371 if (answer != null) {  
372 return answer;  
373 }  
374 }  
375 return defaultValue;  
376 }  
377  
378 /\*\*  
379 \* Looks up the given key in the given map, converting the result into  
380 \* a string, using the default value if the conversion fails.  
381 \*  
382 \* @param <K> the key type  
383 \* @param map the map whose value to look up  
384 \* @param key the key of the value to look up in that map  
385 \* @param defaultValue what to return if the value is null or if the  
386 \* conversion fails  
387 \* @return the value in the map as a string, or defaultValue if the  
388 \* original value is null, the map is null or the string conversion fails  
389 \*/  
390 public static <K> String getString(final Map<? super K, ?> map, final K key, final String defaultValue) {  
391 String answer = getString(map, key);  
392 if (answer == null) {  
393 answer = defaultValue;  
394 }  
395 return answer;  
396 }  
397  
398 /\*\*  
399 \* Looks up the given key in the given map, converting the result into  
400 \* a boolean, using the default value if the conversion fails.  
401 \*  
402 \* @param <K> the key type  
403 \* @param map the map whose value to look up  
404 \* @param key the key of the value to look up in that map  
405 \* @param defaultValue what to return if the value is null or if the  
406 \* conversion fails  
407 \* @return the value in the map as a boolean, or defaultValue if the  
408 \* original value is null, the map is null or the boolean conversion fails  
409 \*/  
410 public static <K> Boolean getBoolean(final Map<? super K, ?> map, final K key, final Boolean defaultValue) {  
411 Boolean answer = getBoolean(map, key);  
412 if (answer == null) {  
413 answer = defaultValue;  
414 }  
415 return answer;  
416 }  
417  
418 /\*\*  
419 \* Looks up the given key in the given map, converting the result into  
420 \* a number, using the default value if the conversion fails.  
421 \*  
422 \* @param <K> the key type  
423 \* @param map the map whose value to look up  
424 \* @param key the key of the value to look up in that map  
425 \* @param defaultValue what to return if the value is null or if the  
426 \* conversion fails  
427 \* @return the value in the map as a number, or defaultValue if the  
428 \* original value is null, the map is null or the number conversion fails  
429 \*/  
430 public static <K> Number getNumber(final Map<? super K, ?> map, final K key, final Number defaultValue) {  
431 Number answer = getNumber(map, key);  
432 if (answer == null) {  
433 answer = defaultValue;  
434 }  
435 return answer;  
436 }  
437  
438 /\*\*  
439 \* Looks up the given key in the given map, converting the result into  
440 \* a byte, using the default value if the conversion fails.  
441 \*  
442 \* @param <K> the key type  
443 \* @param map the map whose value to look up  
444 \* @param key the key of the value to look up in that map  
445 \* @param defaultValue what to return if the value is null or if the  
446 \* conversion fails  
447 \* @return the value in the map as a number, or defaultValue if the  
448 \* original value is null, the map is null or the number conversion fails  
449 \*/  
450 public static <K> Byte getByte(final Map<? super K, ?> map, final K key, final Byte defaultValue) {  
451 Byte answer = getByte(map, key);  
452 if (answer == null) {  
453 answer = defaultValue;  
454 }  
455 return answer;  
456 }  
457  
458 /\*\*  
459 \* Looks up the given key in the given map, converting the result into  
460 \* a short, using the default value if the conversion fails.  
461 \*  
462 \* @param <K> the key type  
463 \* @param map the map whose value to look up  
464 \* @param key the key of the value to look up in that map  
465 \* @param defaultValue what to return if the value is null or if the  
466 \* conversion fails  
467 \* @return the value in the map as a number, or defaultValue if the  
468 \* original value is null, the map is null or the number conversion fails  
469 \*/  
470 public static <K> Short getShort(final Map<? super K, ?> map, final K key, final Short defaultValue) {  
471 Short answer = getShort(map, key);  
472 if (answer == null) {  
473 answer = defaultValue;  
474 }  
475 return answer;  
476 }  
477  
478 /\*\*  
479 \* Looks up the given key in the given map, converting the result into  
480 \* an integer, using the default value if the conversion fails.  
481 \*  
482 \* @param <K> the key type  
483 \* @param map the map whose value to look up  
484 \* @param key the key of the value to look up in that map  
485 \* @param defaultValue what to return if the value is null or if the  
486 \* conversion fails  
487 \* @return the value in the map as a number, or defaultValue if the  
488 \* original value is null, the map is null or the number conversion fails  
489 \*/  
490 public static <K> Integer getInteger(final Map<? super K, ?> map, final K key, final Integer defaultValue) {  
491 Integer answer = getInteger(map, key);  
492 if (answer == null) {  
493 answer = defaultValue;  
494 }  
495 return answer;  
496 }  
497  
498 /\*\*  
499 \* Looks up the given key in the given map, converting the result into  
500 \* a long, using the default value if the conversion fails.  
501 \*  
502 \* @param <K> the key type  
503 \* @param map the map whose value to look up  
504 \* @param key the key of the value to look up in that map  
505 \* @param defaultValue what to return if the value is null or if the  
506 \* conversion fails  
507 \* @return the value in the map as a number, or defaultValue if the  
508 \* original value is null, the map is null or the number conversion fails  
509 \*/  
510 public static <K> Long getLong(final Map<? super K, ?> map, final K key, final Long defaultValue) {  
511 Long answer = getLong(map, key);  
512 if (answer == null) {  
513 answer = defaultValue;  
514 }  
515 return answer;  
516 }  
517  
518 /\*\*  
519 \* Looks up the given key in the given map, converting the result into  
520 \* a float, using the default value if the conversion fails.  
521 \*  
522 \* @param <K> the key type  
523 \* @param map the map whose value to look up  
524 \* @param key the key of the value to look up in that map  
525 \* @param defaultValue what to return if the value is null or if the  
526 \* conversion fails  
527 \* @return the value in the map as a number, or defaultValue if the  
528 \* original value is null, the map is null or the number conversion fails  
529 \*/  
530 public static <K> Float getFloat(final Map<? super K, ?> map, final K key, final Float defaultValue) {  
531 Float answer = getFloat(map, key);  
532 if (answer == null) {  
533 answer = defaultValue;  
534 }  
535 return answer;  
536 }  
537  
538 /\*\*  
539 \* Looks up the given key in the given map, converting the result into  
540 \* a double, using the default value if the conversion fails.  
541 \*  
542 \* @param <K> the key type  
543 \* @param map the map whose value to look up  
544 \* @param key the key of the value to look up in that map  
545 \* @param defaultValue what to return if the value is null or if the  
546 \* conversion fails  
547 \* @return the value in the map as a number, or defaultValue if the  
548 \* original value is null, the map is null or the number conversion fails  
549 \*/  
550 public static <K> Double getDouble(final Map<? super K, ?> map, final K key, final Double defaultValue) {  
551 Double answer = getDouble(map, key);  
552 if (answer == null) {  
553 answer = defaultValue;  
554 }  
555 return answer;  
556 }  
557  
558 /\*\*  
559 \* Looks up the given key in the given map, converting the result into  
560 \* a map, using the default value if the conversion fails.  
561 \*  
562 \* @param <K> the key type  
563 \* @param map the map whose value to look up  
564 \* @param key the key of the value to look up in that map  
565 \* @param defaultValue what to return if the value is null or if the  
566 \* conversion fails  
567 \* @return the value in the map as a number, or defaultValue if the  
568 \* original value is null, the map is null or the map conversion fails  
569 \*/  
570 public static <K> Map<?, ?> getMap(final Map<? super K, ?> map, final K key, final Map<?, ?> defaultValue) {  
571 Map<?, ?> answer = getMap(map, key);  
572 if (answer == null) {  
573 answer = defaultValue;  
574 }  
575 return answer;  
576 }  
577  
578 // Type safe primitive getters  
579 //-------------------------------------------------------------------------  
580 /\*\*  
581 \* Gets a boolean from a Map in a null-safe manner.  
582 \* <p>  
583 \* If the value is a <code>Boolean</code> its value is returned.  
584 \* If the value is a <code>String</code> and it equals 'true' ignoring case  
585 \* then <code>true</code> is returned, otherwise <code>false</code>.  
586 \* If the value is a <code>Number</code> an integer zero value returns  
587 \* <code>false</code> and non-zero returns <code>true</code>.  
588 \* Otherwise, <code>false</code> is returned.  
589 \*  
590 \* @param <K> the key type  
591 \* @param map the map to use  
592 \* @param key the key to look up  
593 \* @return the value in the Map as a Boolean, <code>false</code> if null map input  
594 \*/  
595 public static <K> boolean getBooleanValue(final Map<? super K, ?> map, final K key) {  
596 return Boolean.TRUE.equals(getBoolean(map, key));  
597 }  
598  
599 /\*\*  
600 \* Gets a byte from a Map in a null-safe manner.  
601 \* <p>  
602 \* The byte is obtained from the results of {@link #getNumber(Map,Object)}.  
603 \*  
604 \* @param <K> the key type  
605 \* @param map the map to use  
606 \* @param key the key to look up  
607 \* @return the value in the Map as a byte, <code>0</code> if null map input  
608 \*/  
609 public static <K> byte getByteValue(final Map<? super K, ?> map, final K key) {  
610 final Byte byteObject = getByte(map, key);  
611 if (byteObject == null) {  
612 return 0;  
613 }  
614 return byteObject.byteValue();  
615 }  
616  
617 /\*\*  
618 \* Gets a short from a Map in a null-safe manner.  
619 \* <p>  
620 \* The short is obtained from the results of {@link #getNumber(Map,Object)}.  
621 \*  
622 \* @param <K> the key type  
623 \* @param map the map to use  
624 \* @param key the key to look up  
625 \* @return the value in the Map as a short, <code>0</code> if null map input  
626 \*/  
627 public static <K> short getShortValue(final Map<? super K, ?> map, final K key) {  
628 final Short shortObject = getShort(map, key);  
629 if (shortObject == null) {  
630 return 0;  
631 }  
632 return shortObject.shortValue();  
633 }  
634  
635 /\*\*  
636 \* Gets an int from a Map in a null-safe manner.  
637 \* <p>  
638 \* The int is obtained from the results of {@link #getNumber(Map,Object)}.  
639 \*  
640 \* @param <K> the key type  
641 \* @param map the map to use  
642 \* @param key the key to look up  
643 \* @return the value in the Map as an int, <code>0</code> if null map input  
644 \*/  
645 public static <K> int getIntValue(final Map<? super K, ?> map, final K key) {  
646 final Integer integerObject = getInteger(map, key);  
647 if (integerObject == null) {  
648 return 0;  
649 }  
650 return integerObject.intValue();  
651 }  
652  
653 /\*\*  
654 \* Gets a long from a Map in a null-safe manner.  
655 \* <p>  
656 \* The long is obtained from the results of {@link #getNumber(Map,Object)}.  
657 \*  
658 \* @param <K> the key type  
659 \* @param map the map to use  
660 \* @param key the key to look up  
661 \* @return the value in the Map as a long, <code>0L</code> if null map input  
662 \*/  
663 public static <K> long getLongValue(final Map<? super K, ?> map, final K key) {  
664 final Long longObject = getLong(map, key);  
665 if (longObject == null) {  
666 return 0L;  
667 }  
668 return longObject.longValue();  
669 }  
670  
671 /\*\*  
672 \* Gets a float from a Map in a null-safe manner.  
673 \* <p>  
674 \* The float is obtained from the results of {@link #getNumber(Map,Object)}.  
675 \*  
676 \* @param <K> the key type  
677 \* @param map the map to use  
678 \* @param key the key to look up  
679 \* @return the value in the Map as a float, <code>0.0F</code> if null map input  
680 \*/  
681 public static <K> float getFloatValue(final Map<? super K, ?> map, final K key) {  
682 final Float floatObject = getFloat(map, key);  
683 if (floatObject == null) {  
684 return 0f;  
685 }  
686 return floatObject.floatValue();  
687 }  
688  
689 /\*\*  
690 \* Gets a double from a Map in a null-safe manner.  
691 \* <p>  
692 \* The double is obtained from the results of {@link #getNumber(Map,Object)}.  
693 \*  
694 \* @param <K> the key type  
695 \* @param map the map to use  
696 \* @param key the key to look up  
697 \* @return the value in the Map as a double, <code>0.0</code> if null map input  
698 \*/  
699 public static <K> double getDoubleValue(final Map<? super K, ?> map, final K key) {  
700 final Double doubleObject = getDouble(map, key);  
701 if (doubleObject == null) {  
702 return 0d;  
703 }  
704 return doubleObject.doubleValue();  
705 }  
706  
707 // Type safe primitive getters with default values  
708 //-------------------------------------------------------------------------  
709 /\*\*  
710 \* Gets a boolean from a Map in a null-safe manner,  
711 \* using the default value if the conversion fails.  
712 \* <p>  
713 \* If the value is a <code>Boolean</code> its value is returned.  
714 \* If the value is a <code>String</code> and it equals 'true' ignoring case  
715 \* then <code>true</code> is returned, otherwise <code>false</code>.  
716 \* If the value is a <code>Number</code> an integer zero value returns  
717 \* <code>false</code> and non-zero returns <code>true</code>.  
718 \* Otherwise, <code>defaultValue</code> is returned.  
719 \*  
720 \* @param <K> the key type  
721 \* @param map the map to use  
722 \* @param key the key to look up  
723 \* @param defaultValue return if the value is null or if the conversion fails  
724 \* @return the value in the Map as a Boolean, <code>defaultValue</code> if null map input  
725 \*/  
726 public static <K> boolean getBooleanValue(final Map<? super K, ?> map, final K key, final boolean defaultValue) {  
727 final Boolean booleanObject = getBoolean(map, key);  
728 if (booleanObject == null) {  
729 return defaultValue;  
730 }  
731 return booleanObject.booleanValue();  
732 }  
733  
734 /\*\*  
735 \* Gets a byte from a Map in a null-safe manner,  
736 \* using the default value if the conversion fails.  
737 \* <p>  
738 \* The byte is obtained from the results of {@link #getNumber(Map,Object)}.  
739 \*  
740 \* @param <K> the key type  
741 \* @param map the map to use  
742 \* @param key the key to look up  
743 \* @param defaultValue return if the value is null or if the conversion fails  
744 \* @return the value in the Map as a byte, <code>defaultValue</code> if null map input  
745 \*/  
746 public static <K> byte getByteValue(final Map<? super K, ?> map, final K key, final byte defaultValue) {  
747 final Byte byteObject = getByte(map, key);  
748 if (byteObject == null) {  
749 return defaultValue;  
750 }  
751 return byteObject.byteValue();  
752 }  
753  
754 /\*\*  
755 \* Gets a short from a Map in a null-safe manner,  
756 \* using the default value if the conversion fails.  
757 \* <p>  
758 \* The short is obtained from the results of {@link #getNumber(Map,Object)}.  
759 \*  
760 \* @param <K> the key type  
761 \* @param map the map to use  
762 \* @param key the key to look up  
763 \* @param defaultValue return if the value is null or if the conversion fails  
764 \* @return the value in the Map as a short, <code>defaultValue</code> if null map input  
765 \*/  
766 public static <K> short getShortValue(final Map<? super K, ?> map, final K key, final short defaultValue) {  
767 final Short shortObject = getShort(map, key);  
768 if (shortObject == null) {  
769 return defaultValue;  
770 }  
771 return shortObject.shortValue();  
772 }  
773  
774 /\*\*  
775 \* Gets an int from a Map in a null-safe manner,  
776 \* using the default value if the conversion fails.  
777 \* <p>  
778 \* The int is obtained from the results of {@link #getNumber(Map,Object)}.  
779 \*  
780 \* @param <K> the key type  
781 \* @param map the map to use  
782 \* @param key the key to look up  
783 \* @param defaultValue return if the value is null or if the conversion fails  
784 \* @return the value in the Map as an int, <code>defaultValue</code> if null map input  
785 \*/  
786 public static <K> int getIntValue(final Map<? super K, ?> map, final K key, final int defaultValue) {  
787 final Integer integerObject = getInteger(map, key);  
788 if (integerObject == null) {  
789 return defaultValue;  
790 }  
791 return integerObject.intValue();  
792 }  
793  
794 /\*\*  
795 \* Gets a long from a Map in a null-safe manner,  
796 \* using the default value if the conversion fails.  
797 \* <p>  
798 \* The long is obtained from the results of {@link #getNumber(Map,Object)}.  
799 \*  
800 \* @param <K> the key type  
801 \* @param map the map to use  
802 \* @param key the key to look up  
803 \* @param defaultValue return if the value is null or if the conversion fails  
804 \* @return the value in the Map as a long, <code>defaultValue</code> if null map input  
805 \*/  
806 public static <K> long getLongValue(final Map<? super K, ?> map, final K key, final long defaultValue) {  
807 final Long longObject = getLong(map, key);  
808 if (longObject == null) {  
809 return defaultValue;  
810 }  
811 return longObject.longValue();  
812 }  
813  
814 /\*\*  
815 \* Gets a float from a Map in a null-safe manner,  
816 \* using the default value if the conversion fails.  
817 \* <p>  
818 \* The float is obtained from the results of {@link #getNumber(Map,Object)}.  
819 \*  
820 \* @param <K> the key type  
821 \* @param map the map to use  
822 \* @param key the key to look up  
823 \* @param defaultValue return if the value is null or if the conversion fails  
824 \* @return the value in the Map as a float, <code>defaultValue</code> if null map input  
825 \*/  
826 public static <K> float getFloatValue(final Map<? super K, ?> map, final K key, final float defaultValue) {  
827 final Float floatObject = getFloat(map, key);  
828 if (floatObject == null) {  
829 return defaultValue;  
830 }  
831 return floatObject.floatValue();  
832 }  
833  
834 /\*\*  
835 \* Gets a double from a Map in a null-safe manner,  
836 \* using the default value if the conversion fails.  
837 \* <p>  
838 \* The double is obtained from the results of {@link #getNumber(Map,Object)}.  
839 \*  
840 \* @param <K> the key type  
841 \* @param map the map to use  
842 \* @param key the key to look up  
843 \* @param defaultValue return if the value is null or if the conversion fails  
844 \* @return the value in the Map as a double, <code>defaultValue</code> if null map input  
845 \*/  
846 public static <K> double getDoubleValue(final Map<? super K, ?> map, final K key, final double defaultValue) {  
847 final Double doubleObject = getDouble(map, key);  
848 if (doubleObject == null) {  
849 return defaultValue;  
850 }  
851 return doubleObject.doubleValue();  
852 }  
853  
854 // Conversion methods  
855 //-------------------------------------------------------------------------  
856 /\*\*  
857 \* Gets a new Properties object initialised with the values from a Map.  
858 \* A null input will return an empty properties object.  
859 \* <p>  
860 \* A Properties object may only store non-null keys and values, thus if  
861 \* the provided map contains either a key or value which is {@code null},  
862 \* a {@link NullPointerException} will be thrown.  
863 \*  
864 \* @param <K> the key type  
865 \* @param <V> the value type  
866 \* @param map the map to convert to a Properties object  
867 \* @return the properties object  
868 \* @throws NullPointerException if a key or value in the provided map is {@code null}  
869 \*/  
870 public static <K, V> Properties toProperties(final Map<K, V> map) {  
871 final Properties answer = new Properties();  
872 if (map != null) {  
873 for (final Entry<K, V> entry2 : map.entrySet()) {  
874 final Map.Entry<?, ?> entry = entry2;  
875 final Object key = entry.getKey();  
876 final Object value = entry.getValue();  
877 answer.put(key, value);  
878 }  
879 }  
880 return answer;  
881 }  
882  
883 /\*\*  
884 \* Creates a new HashMap using data copied from a ResourceBundle.  
885 \*  
886 \* @param resourceBundle the resource bundle to convert, may not be null  
887 \* @return the hashmap containing the data  
888 \* @throws NullPointerException if the bundle is null  
889 \*/  
890 public static Map<String, Object> toMap(final ResourceBundle resourceBundle) {  
891 final Enumeration<String> enumeration = resourceBundle.getKeys();  
892 final Map<String, Object> map = new HashMap<>();  
893  
894 while (enumeration.hasMoreElements()) {  
895 final String key = enumeration.nextElement();  
896 final Object value = resourceBundle.getObject(key);  
897 map.put(key, value);  
898 }  
899  
900 return map;  
901 }  
902  
903 // Printing methods  
904 //-------------------------------------------------------------------------  
905 /\*\*  
906 \* Prints the given map with nice line breaks.  
907 \* <p>  
908 \* This method prints a nicely formatted String describing the Map.  
909 \* Each map entry will be printed with key and value.  
910 \* When the value is a Map, recursive behaviour occurs.  
911 \* <p>  
912 \* This method is NOT thread-safe in any special way. You must manually  
913 \* synchronize on either this class or the stream as required.  
914 \*  
915 \* @param out the stream to print to, must not be null  
916 \* @param label The label to be used, may be <code>null</code>.  
917 \* If <code>null</code>, the label is not output.  
918 \* It typically represents the name of the property in a bean or similar.  
919 \* @param map The map to print, may be <code>null</code>.  
920 \* If <code>null</code>, the text 'null' is output.  
921 \* @throws NullPointerException if the stream is <code>null</code>  
922 \*/  
923 public static void verbosePrint(final PrintStream out, final Object label, final Map<?, ?> map) {  
924 verbosePrintInternal(out, label, map, new ArrayDeque<Map<?, ?>>(), false);  
925 }  
926  
927 /\*\*  
928 \* Prints the given map with nice line breaks.  
929 \* <p>  
930 \* This method prints a nicely formatted String describing the Map.  
931 \* Each map entry will be printed with key, value and value classname.  
932 \* When the value is a Map, recursive behaviour occurs.  
933 \* <p>  
934 \* This method is NOT thread-safe in any special way. You must manually  
935 \* synchronize on either this class or the stream as required.  
936 \*  
937 \* @param out the stream to print to, must not be null  
938 \* @param label The label to be used, may be <code>null</code>.  
939 \* If <code>null</code>, the label is not output.  
940 \* It typically represents the name of the property in a bean or similar.  
941 \* @param map The map to print, may be <code>null</code>.  
942 \* If <code>null</code>, the text 'null' is output.  
943 \* @throws NullPointerException if the stream is <code>null</code>  
944 \*/  
945 public static void debugPrint(final PrintStream out, final Object label, final Map<?, ?> map) {  
946 verbosePrintInternal(out, label, map, new ArrayDeque<Map<?, ?>>(), true);  
947 }  
948  
949 // Implementation methods  
950 //-------------------------------------------------------------------------  
951 /\*\*  
952 \* Implementation providing functionality for {@link #debugPrint} and for  
953 \* {@link #verbosePrint}. This prints the given map with nice line breaks.  
954 \* If the debug flag is true, it additionally prints the type of the object  
955 \* value. If the contents of a map include the map itself, then the text  
956 \* <em>(this Map)</em> is printed out. If the contents include a  
957 \* parent container of the map, the text <em>(ancestor[i] Map)</em> is  
958 \* printed, where i actually indicates the number of levels which must be  
959 \* traversed in the sequential list of ancestors (e.g. father, grandfather,  
960 \* great-grandfather, etc).  
961 \*  
962 \* @param out the stream to print to  
963 \* @param label the label to be used, may be <code>null</code>.  
964 \* If <code>null</code>, the label is not output.  
965 \* It typically represents the name of the property in a bean or similar.  
966 \* @param map the map to print, may be <code>null</code>.  
967 \* If <code>null</code>, the text 'null' is output  
968 \* @param lineage a stack consisting of any maps in which the previous  
969 \* argument is contained. This is checked to avoid infinite recursion when  
970 \* printing the output  
971 \* @param debug flag indicating whether type names should be output.  
972 \* @throws NullPointerException if the stream is <code>null</code>  
973 \*/  
974 private static void verbosePrintInternal(final PrintStream out, final Object label, final Map<?, ?> map,  
975 final Deque<Map<?, ?>> lineage, final boolean debug) {  
976 printIndent(out, lineage.size());  
977  
978 if (map == null) {  
979 if (label != null) {  
980 out.print(label);  
981 out.print(" = ");  
982 }  
983 out.println("null");  
984 return;  
985 }  
986 if (label != null) {  
987 out.print(label);  
988 out.println(" = ");  
989 }  
990  
991 printIndent(out, lineage.size());  
992 out.println("{");  
993  
994 lineage.addLast(map);  
995  
996 for (final Map.Entry<?, ?> entry : map.entrySet()) {  
997 final Object childKey = entry.getKey();  
998 final Object childValue = entry.getValue();  
999 if (childValue instanceof Map && !lineage.contains(childValue)) {  
1000 verbosePrintInternal(  
1001 out,  
1002 childKey == null ? "null" : childKey,  
1003 (Map<?, ?>) childValue,  
1004 lineage,  
1005 debug);  
1006 } else {  
1007 printIndent(out, lineage.size());  
1008 out.print(childKey);  
1009 out.print(" = ");  
1010  
1011 final int lineageIndex =  
1012 IterableUtils.indexOf(lineage,  
1013 PredicateUtils.equalPredicate(childValue));  
1014 if (lineageIndex == -1) {  
1015 out.print(childValue);  
1016 } else if (lineage.size() - 1 == lineageIndex) {  
1017 out.print("(this Map)");  
1018 } else {  
1019 out.print(  
1020 "(ancestor["  
1021 + (lineage.size() - 1 - lineageIndex - 1)  
1022 + "] Map)");  
1023 }  
1024  
1025 if (debug && childValue != null) {  
1026 out.print(' ');  
1027 out.println(childValue.getClass().getName());  
1028 } else {  
1029 out.println();  
1030 }  
1031 }  
1032 }  
1033  
1034 lineage.removeLast();  
1035  
1036 printIndent(out, lineage.size());  
1037 out.println(debug ? "} " + map.getClass().getName() : "}");  
1038 }  
1039  
1040 /\*\*  
1041 \* Writes indentation to the given stream.  
1042 \*  
1043 \* @param out the stream to indent  
1044 \*/  
1045 private static void printIndent(final PrintStream out, final int indent) {  
1046 for (int i = 0; i < indent; i++) {  
1047 out.print(INDENT\_STRING);  
1048 }  
1049 }  
1050  
1051 // Misc  
1052 //-----------------------------------------------------------------------  
1053 /\*\*  
1054 \* Inverts the supplied map returning a new HashMap such that the keys of  
1055 \* the input are swapped with the values.  
1056 \* <p>  
1057 \* This operation assumes that the inverse mapping is well defined.  
1058 \* If the input map had multiple entries with the same value mapped to  
1059 \* different keys, the returned map will map one of those keys to the  
1060 \* value, but the exact key which will be mapped is undefined.  
1061 \*  
1062 \* @param <K> the key type  
1063 \* @param <V> the value type  
1064 \* @param map the map to invert, may not be null  
1065 \* @return a new HashMap containing the inverted data  
1066 \* @throws NullPointerException if the map is null  
1067 \*/  
1068 public static <K, V> Map<V, K> invertMap(final Map<K, V> map) {  
1069 final Map<V, K> out = new HashMap<>(map.size());  
1070 for (final Entry<K, V> entry : map.entrySet()) {  
1071 out.put(entry.getValue(), entry.getKey());  
1072 }  
1073 return out;  
1074 }  
1075  
1076 //-----------------------------------------------------------------------  
1077 /\*\*  
1078 \* Protects against adding null values to a map.  
1079 \* <p>  
1080 \* This method checks the value being added to the map, and if it is null  
1081 \* it is replaced by an empty string.  
1082 \* <p>  
1083 \* This could be useful if the map does not accept null values, or for  
1084 \* receiving data from a source that may provide null or empty string  
1085 \* which should be held in the same way in the map.  
1086 \* <p>  
1087 \* Keys are not validated.  
1088 \* Note that this method can be used to circumvent the map's  
1089 \* value type at runtime.  
1090 \*  
1091 \* @param <K> the key type  
1092 \* @param map the map to add to, may not be null  
1093 \* @param key the key  
1094 \* @param value the value, null converted to ""  
1095 \* @throws NullPointerException if the map is null  
1096 \*/  
1097 public static <K> void safeAddToMap(final Map<? super K, Object> map, final K key, final Object value)  
1098 throws NullPointerException {  
1099 map.put(key, value == null ? "" : value);  
1100 }  
1101  
1102 //-----------------------------------------------------------------------  
1103 /\*\*  
1104 \* Puts all the keys and values from the specified array into the map.  
1105 \* <p>  
1106 \* This method is an alternative to the {@link java.util.Map#putAll(java.util.Map)}  
1107 \* method and constructors. It allows you to build a map from an object array  
1108 \* of various possible styles.  
1109 \* <p>  
1110 \* If the first entry in the object array implements {@link java.util.Map.Entry}  
1111 \* or {@link KeyValue} then the key and value are added from that object.  
1112 \* If the first entry in the object array is an object array itself, then  
1113 \* it is assumed that index 0 in the sub-array is the key and index 1 is the value.  
1114 \* Otherwise, the array is treated as keys and values in alternate indices.  
1115 \* <p>  
1116 \* For example, to create a color map:  
1117 \* <pre>  
1118 \* Map colorMap = MapUtils.putAll(new HashMap(), new String[][] {  
1119 \* {"RED", "#FF0000"},  
1120 \* {"GREEN", "#00FF00"},  
1121 \* {"BLUE", "#0000FF"}  
1122 \* });  
1123 \* </pre>  
1124 \* or:  
1125 \* <pre>  
1126 \* Map colorMap = MapUtils.putAll(new HashMap(), new String[] {  
1127 \* "RED", "#FF0000",  
1128 \* "GREEN", "#00FF00",  
1129 \* "BLUE", "#0000FF"  
1130 \* });  
1131 \* </pre>  
1132 \* or:  
1133 \* <pre>  
1134 \* Map colorMap = MapUtils.putAll(new HashMap(), new Map.Entry[] {  
1135 \* new DefaultMapEntry("RED", "#FF0000"),  
1136 \* new DefaultMapEntry("GREEN", "#00FF00"),  
1137 \* new DefaultMapEntry("BLUE", "#0000FF")  
1138 \* });  
1139 \* </pre>  
1140 \*  
1141 \* @param <K> the key type  
1142 \* @param <V> the value type  
1143 \* @param map the map to populate, must not be null  
1144 \* @param array an array to populate from, null ignored  
1145 \* @return the input map  
1146 \* @throws NullPointerException if map is null  
1147 \* @throws IllegalArgumentException if sub-array or entry matching used and an entry is invalid  
1148 \* @throws ClassCastException if the array contents is mixed  
1149 \* @since 3.2  
1150 \*/  
1151 @SuppressWarnings("unchecked") // As per Javadoc throws CCE for invalid array contents  
1152 public static <K, V> Map<K, V> putAll(final Map<K, V> map, final Object[] array) {  
1153 if (map == null) {  
1154 throw new NullPointerException("The map must not be null");  
1155 }  
1156 if (array == null || array.length == 0) {  
1157 return map;  
1158 }  
1159 final Object obj = array[0];  
1160 if (obj instanceof Map.Entry) {  
1161 for (final Object element : array) {  
1162 // cast ok here, type is checked above  
1163 final Map.Entry<K, V> entry = (Map.Entry<K, V>) element;  
1164 map.put(entry.getKey(), entry.getValue());  
1165 }  
1166 } else if (obj instanceof KeyValue) {  
1167 for (final Object element : array) {  
1168 // cast ok here, type is checked above  
1169 final KeyValue<K, V> keyval = (KeyValue<K, V>) element;  
1170 map.put(keyval.getKey(), keyval.getValue());  
1171 }  
1172 } else if (obj instanceof Object[]) {  
1173 for (int i = 0; i < array.length; i++) {  
1174 final Object[] sub = (Object[]) array[i];  
1175 if (sub == null || sub.length < 2) {  
1176 throw new IllegalArgumentException("Invalid array element: " + i);  
1177 }  
1178 // these casts can fail if array has incorrect types  
1179 map.put((K) sub[0], (V) sub[1]);  
1180 }  
1181 } else {  
1182 for (int i = 0; i < array.length - 1;) {  
1183 // these casts can fail if array has incorrect types  
1184 map.put((K) array[i++], (V) array[i++]);  
1185 }  
1186 }  
1187 return map;  
1188 }  
1189  
1190 //-----------------------------------------------------------------------  
1191  
1192 /\*\*  
1193 \* Returns an immutable empty map if the argument is <code>null</code>,  
1194 \* or the argument itself otherwise.  
1195 \*  
1196 \* @param <K> the key type  
1197 \* @param <V> the value type  
1198 \* @param map the map, possibly <code>null</code>  
1199 \* @return an empty map if the argument is <code>null</code>  
1200 \*/  
1201 public static <K,V> Map<K,V> emptyIfNull(final Map<K,V> map) {  
1202 return map == null ? Collections.<K,V>emptyMap() : map;  
1203 }  
1204  
1205 /\*\*  
1206 \* Null-safe check if the specified map is empty.  
1207 \* <p>  
1208 \* Null returns true.  
1209 \*  
1210 \* @param map the map to check, may be null  
1211 \* @return true if empty or null  
1212 \* @since 3.2  
1213 \*/  
1214 public static boolean isEmpty(final Map<?,?> map) {  
1215 return map == null || map.isEmpty();  
1216 }  
1217  
1218 /\*\*  
1219 \* Null-safe check if the specified map is not empty.  
1220 \* <p>  
1221 \* Null returns false.  
1222 \*  
1223 \* @param map the map to check, may be null  
1224 \* @return true if non-null and non-empty  
1225 \* @since 3.2  
1226 \*/  
1227 public static boolean isNotEmpty(final Map<?,?> map) {  
1228 return !MapUtils.isEmpty(map);  
1229 }  
1230  
1231 // Map decorators  
1232 //-----------------------------------------------------------------------  
1233 /\*\*  
1234 \* Returns a synchronized map backed by the given map.  
1235 \* <p>  
1236 \* You must manually synchronize on the returned buffer's iterator to  
1237 \* avoid non-deterministic behavior:  
1238 \*  
1239 \* <pre>  
1240 \* Map m = MapUtils.synchronizedMap(myMap);  
1241 \* Set s = m.keySet(); // outside synchronized block  
1242 \* synchronized (m) { // synchronized on MAP!  
1243 \* Iterator i = s.iterator();  
1244 \* while (i.hasNext()) {  
1245 \* process (i.next());  
1246 \* }  
1247 \* }  
1248 \* </pre>  
1249 \*  
1250 \* This method uses the implementation in {@link java.util.Collections Collections}.  
1251 \*  
1252 \* @param <K> the key type  
1253 \* @param <V> the value type  
1254 \* @param map the map to synchronize, must not be null  
1255 \* @return a synchronized map backed by the given map  
1256 \*/  
1257 public static <K, V> Map<K, V> synchronizedMap(final Map<K, V> map) {  
1258 return Collections.synchronizedMap(map);  
1259 }  
1260  
1261 /\*\*  
1262 \* Returns an unmodifiable map backed by the given map.  
1263 \* <p>  
1264 \* This method uses the implementation in the decorators subpackage.  
1265 \*  
1266 \* @param <K> the key type  
1267 \* @param <V> the value type  
1268 \* @param map the map to make unmodifiable, must not be null  
1269 \* @return an unmodifiable map backed by the given map  
1270 \* @throws NullPointerException if the map is null  
1271 \*/  
1272 public static <K, V> Map<K, V> unmodifiableMap(final Map<? extends K, ? extends V> map) {  
1273 return UnmodifiableMap.unmodifiableMap(map);  
1274 }  
1275  
1276 /\*\*  
1277 \* Returns a predicated (validating) map backed by the given map.  
1278 \* <p>  
1279 \* Only objects that pass the tests in the given predicates can be added to the map.  
1280 \* Trying to add an invalid object results in an IllegalArgumentException.  
1281 \* Keys must pass the key predicate, values must pass the value predicate.  
1282 \* It is important not to use the original map after invoking this method,  
1283 \* as it is a backdoor for adding invalid objects.  
1284 \*  
1285 \* @param <K> the key type  
1286 \* @param <V> the value type  
1287 \* @param map the map to predicate, must not be null  
1288 \* @param keyPred the predicate for keys, null means no check  
1289 \* @param valuePred the predicate for values, null means no check  
1290 \* @return a predicated map backed by the given map  
1291 \* @throws NullPointerException if the Map is null  
1292 \*/  
1293 public static <K, V> IterableMap<K, V> predicatedMap(final Map<K, V> map, final Predicate<? super K> keyPred,  
1294 final Predicate<? super V> valuePred) {  
1295 return PredicatedMap.predicatedMap(map, keyPred, valuePred);  
1296 }  
1297  
1298 /\*\*  
1299 \* Returns a transformed map backed by the given map.  
1300 \* <p>  
1301 \* This method returns a new map (decorating the specified map) that  
1302 \* will transform any new entries added to it.  
1303 \* Existing entries in the specified map will not be transformed.  
1304 \* If you want that behaviour, see {@link TransformedMap#transformedMap}.  
1305 \* <p>  
1306 \* Each object is passed through the transformers as it is added to the  
1307 \* Map. It is important not to use the original map after invoking this  
1308 \* method, as it is a backdoor for adding untransformed objects.  
1309 \* <p>  
1310 \* If there are any elements already in the map being decorated, they  
1311 \* are NOT transformed.  
1312 \*  
1313 \* @param <K> the key type  
1314 \* @param <V> the value type  
1315 \* @param map the map to transform, must not be null, typically empty  
1316 \* @param keyTransformer the transformer for the map keys, null means no transformation  
1317 \* @param valueTransformer the transformer for the map values, null means no transformation  
1318 \* @return a transformed map backed by the given map  
1319 \* @throws NullPointerException if the Map is null  
1320 \*/  
1321 public static <K, V> IterableMap<K, V> transformedMap(final Map<K, V> map,  
1322 final Transformer<? super K, ? extends K> keyTransformer,  
1323 final Transformer<? super V, ? extends V> valueTransformer) {  
1324 return TransformedMap.transformingMap(map, keyTransformer, valueTransformer);  
1325 }  
1326  
1327 /\*\*  
1328 \* Returns a fixed-sized map backed by the given map.  
1329 \* Elements may not be added or removed from the returned map, but  
1330 \* existing elements can be changed (for instance, via the  
1331 \* {@link Map#put(Object,Object)} method).  
1332 \*  
1333 \* @param <K> the key type  
1334 \* @param <V> the value type  
1335 \* @param map the map whose size to fix, must not be null  
1336 \* @return a fixed-size map backed by that map  
1337 \* @throws NullPointerException if the Map is null  
1338 \*/  
1339 public static <K, V> IterableMap<K, V> fixedSizeMap(final Map<K, V> map) {  
1340 return FixedSizeMap.fixedSizeMap(map);  
1341 }  
1342  
1343 /\*\*  
1344 \* Returns a "lazy" map whose values will be created on demand.  
1345 \* <p>  
1346 \* When the key passed to the returned map's {@link Map#get(Object)}  
1347 \* method is not present in the map, then the factory will be used  
1348 \* to create a new object and that object will become the value  
1349 \* associated with that key.  
1350 \* <p>  
1351 \* For instance:  
1352 \* <pre>  
1353 \* Factory factory = new Factory() {  
1354 \* public Object create() {  
1355 \* return new Date();  
1356 \* }  
1357 \* }  
1358 \* Map lazyMap = MapUtils.lazyMap(new HashMap(), factory);  
1359 \* Object obj = lazyMap.get("test");  
1360 \* </pre>  
1361 \*  
1362 \* After the above code is executed, <code>obj</code> will contain  
1363 \* a new <code>Date</code> instance. Furthermore, that <code>Date</code>  
1364 \* instance is the value for the <code>"test"</code> key in the map.  
1365 \*  
1366 \* @param <K> the key type  
1367 \* @param <V> the value type  
1368 \* @param map the map to make lazy, must not be null  
1369 \* @param factory the factory for creating new objects, must not be null  
1370 \* @return a lazy map backed by the given map  
1371 \* @throws NullPointerException if the Map or Factory is null  
1372 \*/  
1373 public static <K, V> IterableMap<K, V> lazyMap(final Map<K, V> map, final Factory<? extends V> factory) {  
1374 return LazyMap.lazyMap(map, factory);  
1375 }  
1376  
1377 /\*\*  
1378 \* Returns a "lazy" map whose values will be created on demand.  
1379 \* <p>  
1380 \* When the key passed to the returned map's {@link Map#get(Object)}  
1381 \* method is not present in the map, then the factory will be used  
1382 \* to create a new object and that object will become the value  
1383 \* associated with that key. The factory is a {@link Transformer}  
1384 \* that will be passed the key which it must transform into the value.  
1385 \* <p>  
1386 \* For instance:  
1387 \* <pre>  
1388 \* Transformer factory = new Transformer() {  
1389 \* public Object transform(Object mapKey) {  
1390 \* return new File(mapKey);  
1391 \* }  
1392 \* }  
1393 \* Map lazyMap = MapUtils.lazyMap(new HashMap(), factory);  
1394 \* Object obj = lazyMap.get("C:/dev");  
1395 \* </pre>  
1396 \*  
1397 \* After the above code is executed, <code>obj</code> will contain  
1398 \* a new <code>File</code> instance for the C drive dev directory.  
1399 \* Furthermore, that <code>File</code> instance is the value for the  
1400 \* <code>"C:/dev"</code> key in the map.  
1401 \* <p>  
1402 \* If a lazy map is wrapped by a synchronized map, the result is a simple  
1403 \* synchronized cache. When an object is not is the cache, the cache itself  
1404 \* calls back to the factory Transformer to populate itself, all within the  
1405 \* same synchronized block.  
1406 \*  
1407 \* @param <K> the key type  
1408 \* @param <V> the value type  
1409 \* @param map the map to make lazy, must not be null  
1410 \* @param transformerFactory the factory for creating new objects, must not be null  
1411 \* @return a lazy map backed by the given map  
1412 \* @throws NullPointerException if the Map or Transformer is null  
1413 \*/  
1414 public static <K, V> IterableMap<K, V> lazyMap(final Map<K, V> map,  
1415 final Transformer<? super K, ? extends V> transformerFactory) {  
1416 return LazyMap.lazyMap(map, transformerFactory);  
1417 }  
1418  
1419 /\*\*  
1420 \* Returns a map that maintains the order of keys that are added  
1421 \* backed by the given map.  
1422 \* <p>  
1423 \* If a key is added twice, the order is determined by the first add.  
1424 \* The order is observed through the keySet, values and entrySet.  
1425 \*  
1426 \* @param <K> the key type  
1427 \* @param <V> the value type  
1428 \* @param map the map to order, must not be null  
1429 \* @return an ordered map backed by the given map  
1430 \* @throws NullPointerException if the Map is null  
1431 \*/  
1432 public static <K, V> OrderedMap<K, V> orderedMap(final Map<K, V> map) {  
1433 return ListOrderedMap.listOrderedMap(map);  
1434 }  
1435  
1436 /\*\*  
1437 \* Creates a mult-value map backed by the given map which returns  
1438 \* collections of type ArrayList.  
1439 \*  
1440 \* @param <K> the key type  
1441 \* @param <V> the value type  
1442 \* @param map the map to decorate  
1443 \* @return a multi-value map backed by the given map which returns ArrayLists of values.  
1444 \* @see MultiValueMap  
1445 \* @since 3.2  
1446 \* @deprecated since 4.1, use {@link MultiValuedMap} instead  
1447 \*/  
1448 @Deprecated  
1449 public static <K, V> MultiValueMap<K, V> multiValueMap(final Map<K, ? super Collection<V>> map) {  
1450 return MultiValueMap.<K, V>multiValueMap(map);  
1451 }  
1452  
1453 /\*\*  
1454 \* Creates a multi-value map backed by the given map which returns  
1455 \* collections of the specified type.  
1456 \*  
1457 \* @param <K> the key type  
1458 \* @param <V> the value type  
1459 \* @param <C> the collection class type  
1460 \* @param map the map to decorate  
1461 \* @param collectionClass the type of collections to return from the map  
1462 \* (must contain public no-arg constructor and extend Collection)  
1463 \* @return a multi-value map backed by the given map which returns collections of the specified type  
1464 \* @see MultiValueMap  
1465 \* @since 3.2  
1466 \* @deprecated since 4.1, use {@link MultiValuedMap} instead  
1467 \*/  
1468 @Deprecated  
1469 public static <K, V, C extends Collection<V>> MultiValueMap<K, V> multiValueMap(final Map<K, C> map,  
1470 final Class<C> collectionClass) {  
1471 return MultiValueMap.multiValueMap(map, collectionClass);  
1472 }  
1473  
1474 /\*\*  
1475 \* Creates a multi-value map backed by the given map which returns  
1476 \* collections created by the specified collection factory.  
1477 \*  
1478 \* @param <K> the key type  
1479 \* @param <V> the value type  
1480 \* @param <C> the collection class type  
1481 \* @param map the map to decorate  
1482 \* @param collectionFactory a factor which creates collection objects  
1483 \* @return a multi-value map backed by the given map which returns collections  
1484 \* created by the specified collection factory  
1485 \* @see MultiValueMap  
1486 \* @since 3.2  
1487 \* @deprecated since 4.1, use {@link MultiValuedMap} instead  
1488 \*/  
1489 @Deprecated  
1490 public static <K, V, C extends Collection<V>> MultiValueMap<K, V> multiValueMap(final Map<K, C> map,  
1491 final Factory<C> collectionFactory) {  
1492 return MultiValueMap.multiValueMap(map, collectionFactory);  
1493 }  
1494  
1495 // SortedMap decorators  
1496 //-----------------------------------------------------------------------  
1497 /\*\*  
1498 \* Returns a synchronized sorted map backed by the given sorted map.  
1499 \* <p>  
1500 \* You must manually synchronize on the returned buffer's iterator to  
1501 \* avoid non-deterministic behavior:  
1502 \*  
1503 \* <pre>  
1504 \* Map m = MapUtils.synchronizedSortedMap(myMap);  
1505 \* Set s = m.keySet(); // outside synchronized block  
1506 \* synchronized (m) { // synchronized on MAP!  
1507 \* Iterator i = s.iterator();  
1508 \* while (i.hasNext()) {  
1509 \* process (i.next());  
1510 \* }  
1511 \* }  
1512 \* </pre>  
1513 \*  
1514 \* This method uses the implementation in {@link java.util.Collections Collections}.  
1515 \*  
1516 \* @param <K> the key type  
1517 \* @param <V> the value type  
1518 \* @param map the map to synchronize, must not be null  
1519 \* @return a synchronized map backed by the given map  
1520 \* @throws NullPointerException if the map is null  
1521 \*/  
1522 public static <K, V> SortedMap<K, V> synchronizedSortedMap(final SortedMap<K, V> map) {  
1523 return Collections.synchronizedSortedMap(map);  
1524 }  
1525  
1526 /\*\*  
1527 \* Returns an unmodifiable sorted map backed by the given sorted map.  
1528 \* <p>  
1529 \* This method uses the implementation in the decorators subpackage.  
1530 \*  
1531 \* @param <K> the key type  
1532 \* @param <V> the value type  
1533 \* @param map the sorted map to make unmodifiable, must not be null  
1534 \* @return an unmodifiable map backed by the given map  
1535 \* @throws NullPointerException if the map is null  
1536 \*/  
1537 public static <K, V> SortedMap<K, V> unmodifiableSortedMap(final SortedMap<K, ? extends V> map) {  
1538 return UnmodifiableSortedMap.unmodifiableSortedMap(map);  
1539 }  
1540  
1541 /\*\*  
1542 \* Returns a predicated (validating) sorted map backed by the given map.  
1543 \* <p>  
1544 \* Only objects that pass the tests in the given predicates can be added to the map.  
1545 \* Trying to add an invalid object results in an IllegalArgumentException.  
1546 \* Keys must pass the key predicate, values must pass the value predicate.  
1547 \* It is important not to use the original map after invoking this method,  
1548 \* as it is a backdoor for adding invalid objects.  
1549 \*  
1550 \* @param <K> the key type  
1551 \* @param <V> the value type  
1552 \* @param map the map to predicate, must not be null  
1553 \* @param keyPred the predicate for keys, null means no check  
1554 \* @param valuePred the predicate for values, null means no check  
1555 \* @return a predicated map backed by the given map  
1556 \* @throws NullPointerException if the SortedMap is null  
1557 \*/  
1558 public static <K, V> SortedMap<K, V> predicatedSortedMap(final SortedMap<K, V> map,  
1559 final Predicate<? super K> keyPred, final Predicate<? super V> valuePred) {  
1560 return PredicatedSortedMap.predicatedSortedMap(map, keyPred, valuePred);  
1561 }  
1562  
1563 /\*\*  
1564 \* Returns a transformed sorted map backed by the given map.  
1565 \* <p>  
1566 \* This method returns a new sorted map (decorating the specified map) that  
1567 \* will transform any new entries added to it.  
1568 \* Existing entries in the specified map will not be transformed.  
1569 \* If you want that behaviour, see {@link TransformedSortedMap#transformedSortedMap}.  
1570 \* <p>  
1571 \* Each object is passed through the transformers as it is added to the  
1572 \* Map. It is important not to use the original map after invoking this  
1573 \* method, as it is a backdoor for adding untransformed objects.  
1574 \* <p>  
1575 \* If there are any elements already in the map being decorated, they  
1576 \* are NOT transformed.  
1577 \*  
1578 \* @param <K> the key type  
1579 \* @param <V> the value type  
1580 \* @param map the map to transform, must not be null, typically empty  
1581 \* @param keyTransformer the transformer for the map keys, null means no transformation  
1582 \* @param valueTransformer the transformer for the map values, null means no transformation  
1583 \* @return a transformed map backed by the given map  
1584 \* @throws NullPointerException if the SortedMap is null  
1585 \*/  
1586 public static <K, V> SortedMap<K, V> transformedSortedMap(final SortedMap<K, V> map,  
1587 final Transformer<? super K, ? extends K> keyTransformer,  
1588 final Transformer<? super V, ? extends V> valueTransformer) {  
1589 return TransformedSortedMap.transformingSortedMap(map, keyTransformer, valueTransformer);  
1590 }  
1591  
1592 /\*\*  
1593 \* Returns a fixed-sized sorted map backed by the given sorted map.  
1594 \* Elements may not be added or removed from the returned map, but  
1595 \* existing elements can be changed (for instance, via the  
1596 \* {@link Map#put(Object,Object)} method).  
1597 \*  
1598 \* @param <K> the key type  
1599 \* @param <V> the value type  
1600 \* @param map the map whose size to fix, must not be null  
1601 \* @return a fixed-size map backed by that map  
1602 \* @throws NullPointerException if the SortedMap is null  
1603 \*/  
1604 public static <K, V> SortedMap<K, V> fixedSizeSortedMap(final SortedMap<K, V> map) {  
1605 return FixedSizeSortedMap.fixedSizeSortedMap(map);  
1606 }  
1607  
1608 /\*\*  
1609 \* Returns a "lazy" sorted map whose values will be created on demand.  
1610 \* <p>  
1611 \* When the key passed to the returned map's {@link Map#get(Object)}  
1612 \* method is not present in the map, then the factory will be used  
1613 \* to create a new object and that object will become the value  
1614 \* associated with that key.  
1615 \* <p>  
1616 \* For instance:  
1617 \*  
1618 \* <pre>  
1619 \* Factory factory = new Factory() {  
1620 \* public Object create() {  
1621 \* return new Date();  
1622 \* }  
1623 \* }  
1624 \* SortedMap lazy = MapUtils.lazySortedMap(new TreeMap(), factory);  
1625 \* Object obj = lazy.get("test");  
1626 \* </pre>  
1627 \*  
1628 \* After the above code is executed, <code>obj</code> will contain  
1629 \* a new <code>Date</code> instance. Furthermore, that <code>Date</code>  
1630 \* instance is the value for the <code>"test"</code> key.  
1631 \*  
1632 \* @param <K> the key type  
1633 \* @param <V> the value type  
1634 \* @param map the map to make lazy, must not be null  
1635 \* @param factory the factory for creating new objects, must not be null  
1636 \* @return a lazy map backed by the given map  
1637 \* @throws NullPointerException if the SortedMap or Factory is null  
1638 \*/  
1639 public static <K, V> SortedMap<K, V> lazySortedMap(final SortedMap<K, V> map, final Factory<? extends V> factory) {  
1640 return LazySortedMap.lazySortedMap(map, factory);  
1641 }  
1642  
1643 /\*\*  
1644 \* Returns a "lazy" sorted map whose values will be created on demand.  
1645 \* <p>  
1646 \* When the key passed to the returned map's {@link Map#get(Object)}  
1647 \* method is not present in the map, then the factory will be used  
1648 \* to create a new object and that object will become the value  
1649 \* associated with that key. The factory is a {@link Transformer}  
1650 \* that will be passed the key which it must transform into the value.  
1651 \* <p>  
1652 \* For instance:  
1653 \* <pre>  
1654 \* Transformer factory = new Transformer() {  
1655 \* public Object transform(Object mapKey) {  
1656 \* return new File(mapKey);  
1657 \* }  
1658 \* }  
1659 \* SortedMap lazy = MapUtils.lazySortedMap(new TreeMap(), factory);  
1660 \* Object obj = lazy.get("C:/dev");  
1661 \* </pre>  
1662 \*  
1663 \* After the above code is executed, <code>obj</code> will contain  
1664 \* a new <code>File</code> instance for the C drive dev directory.  
1665 \* Furthermore, that <code>File</code> instance is the value for the  
1666 \* <code>"C:/dev"</code> key in the map.  
1667 \* <p>  
1668 \* If a lazy map is wrapped by a synchronized map, the result is a simple  
1669 \* synchronized cache. When an object is not is the cache, the cache itself  
1670 \* calls back to the factory Transformer to populate itself, all within the  
1671 \* same synchronized block.  
1672 \*  
1673 \* @param <K> the key type  
1674 \* @param <V> the value type  
1675 \* @param map the map to make lazy, must not be null  
1676 \* @param transformerFactory the factory for creating new objects, must not be null  
1677 \* @return a lazy map backed by the given map  
1678 \* @throws NullPointerException if the Map or Transformer is null  
1679 \*/  
1680 public static <K, V> SortedMap<K, V> lazySortedMap(final SortedMap<K, V> map,  
1681 final Transformer<? super K, ? extends V> transformerFactory) {  
1682 return LazySortedMap.lazySortedMap(map, transformerFactory);  
1683 }  
1684  
1685 /\*\*  
1686 \* Populates a Map using the supplied <code>Transformer</code> to transform the elements  
1687 \* into keys, using the unaltered element as the value in the <code>Map</code>.  
1688 \*  
1689 \* @param <K> the key type  
1690 \* @param <V> the value type  
1691 \* @param map the <code>Map</code> to populate.  
1692 \* @param elements the <code>Iterable</code> containing the input values for the map.  
1693 \* @param keyTransformer the <code>Transformer</code> used to transform the element into a key value  
1694 \* @throws NullPointerException if the map, elements or transformer are null  
1695 \*/  
1696 public static <K, V> void populateMap(final Map<K, V> map, final Iterable<? extends V> elements,  
1697 final Transformer<V, K> keyTransformer) {  
1698 populateMap(map, elements, keyTransformer, TransformerUtils.<V>nopTransformer());  
1699 }  
1700  
1701 /\*\*  
1702 \* Populates a Map using the supplied <code>Transformer</code>s to transform the elements  
1703 \* into keys and values.  
1704 \*  
1705 \* @param <K> the key type  
1706 \* @param <V> the value type  
1707 \* @param <E> the type of object contained in the {@link Iterable}  
1708 \* @param map the <code>Map</code> to populate.  
1709 \* @param elements the <code>Iterable</code> containing the input values for the map.  
1710 \* @param keyTransformer the <code>Transformer</code> used to transform the element into a key value  
1711 \* @param valueTransformer the <code>Transformer</code> used to transform the element into a value  
1712 \* @throws NullPointerException if the map, elements or transformers are null  
1713 \*/  
1714 public static <K, V, E> void populateMap(final Map<K, V> map, final Iterable<? extends E> elements,  
1715 final Transformer<E, K> keyTransformer,  
1716 final Transformer<E, V> valueTransformer) {  
1717 final Iterator<? extends E> iter = elements.iterator();  
1718 while (iter.hasNext()) {  
1719 final E temp = iter.next();  
1720 map.put(keyTransformer.transform(temp), valueTransformer.transform(temp));  
1721 }  
1722 }  
1723  
1724 /\*\*  
1725 \* Populates a MultiMap using the supplied <code>Transformer</code> to transform the elements  
1726 \* into keys, using the unaltered element as the value in the <code>MultiMap</code>.  
1727 \*  
1728 \* @param <K> the key type  
1729 \* @param <V> the value type  
1730 \* @param map the <code>MultiMap</code> to populate.  
1731 \* @param elements the <code>Iterable</code> to use as input values for the map.  
1732 \* @param keyTransformer the <code>Transformer</code> used to transform the element into a key value  
1733 \* @throws NullPointerException if the map, elements or transformer are null  
1734 \*/  
1735 public static <K, V> void populateMap(final MultiMap<K, V> map, final Iterable<? extends V> elements,  
1736 final Transformer<V, K> keyTransformer) {  
1737 populateMap(map, elements, keyTransformer, TransformerUtils.<V>nopTransformer());  
1738 }  
1739  
1740 /\*\*  
1741 \* Populates a MultiMap using the supplied <code>Transformer</code>s to transform the elements  
1742 \* into keys and values.  
1743 \*  
1744 \* @param <K> the key type  
1745 \* @param <V> the value type  
1746 \* @param <E> the type of object contained in the {@link Iterable}  
1747 \* @param map the <code>MultiMap</code> to populate.  
1748 \* @param elements the <code>Iterable</code> containing the input values for the map.  
1749 \* @param keyTransformer the <code>Transformer</code> used to transform the element into a key value  
1750 \* @param valueTransformer the <code>Transformer</code> used to transform the element into a value  
1751 \* @throws NullPointerException if the map, collection or transformers are null  
1752 \*/  
1753 public static <K, V, E> void populateMap(final MultiMap<K, V> map, final Iterable<? extends E> elements,  
1754 final Transformer<E, K> keyTransformer,  
1755 final Transformer<E, V> valueTransformer) {  
1756 final Iterator<? extends E> iter = elements.iterator();  
1757 while (iter.hasNext()) {  
1758 final E temp = iter.next();  
1759 map.put(keyTransformer.transform(temp), valueTransformer.transform(temp));  
1760 }  
1761 }  
1762  
1763 /\*\*  
1764 \* Get the specified {@link Map} as an {@link IterableMap}.  
1765 \*  
1766 \* @param <K> the key type  
1767 \* @param <V> the value type  
1768 \* @param map to wrap if necessary.  
1769 \* @return IterableMap<K, V>  
1770 \* @throws NullPointerException if map is null  
1771 \* @since 4.0  
1772 \*/  
1773 public static <K, V> IterableMap<K, V> iterableMap(final Map<K, V> map) {  
1774 if (map == null) {  
1775 throw new NullPointerException("Map must not be null");  
1776 }  
1777 return map instanceof IterableMap ? (IterableMap<K, V>) map : new AbstractMapDecorator<K, V>(map) {};  
1778 }  
1779  
1780 /\*\*  
1781 \* Get the specified {@link SortedMap} as an {@link IterableSortedMap}.  
1782 \*  
1783 \* @param <K> the key type  
1784 \* @param <V> the value type  
1785 \* @param sortedMap to wrap if necessary  
1786 \* @return {@link IterableSortedMap}<K, V>  
1787 \* @throws NullPointerException if sortedMap is null  
1788 \* @since 4.0  
1789 \*/  
1790 public static <K, V> IterableSortedMap<K, V> iterableSortedMap(final SortedMap<K, V> sortedMap) {  
1791 if (sortedMap == null) {  
1792 throw new NullPointerException("Map must not be null");  
1793 }  
1794 return sortedMap instanceof IterableSortedMap ? (IterableSortedMap<K, V>) sortedMap :  
1795 new AbstractSortedMapDecorator<K, V>(sortedMap) {};  
1796 }  
1797  
1798 /\*\*  
1799 \* Gets the given map size or 0 if the map is null  
1800 \* @param map a Map or null  
1801 \* @return the given map size or 0 if the map is null  
1802 \*/  
1803 public static int size(final Map<?, ?> map) {  
1804 return map == null ? 0 : map.size();  
1805 }  
1806  
1807}