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.util.AbstractSet;  
020import java.util.Arrays;  
021import java.util.Collection;  
022import java.util.Collections;  
023import java.util.HashSet;  
024import java.util.IdentityHashMap;  
025import java.util.Iterator;  
026import java.util.NavigableSet;  
027import java.util.Set;  
028import java.util.SortedSet;  
029import java.util.TreeSet;  
030  
031import org.apache.commons.collections4.set.ListOrderedSet;  
032import org.apache.commons.collections4.set.PredicatedNavigableSet;  
033import org.apache.commons.collections4.set.PredicatedSet;  
034import org.apache.commons.collections4.set.PredicatedSortedSet;  
035import org.apache.commons.collections4.set.TransformedNavigableSet;  
036import org.apache.commons.collections4.set.TransformedSet;  
037import org.apache.commons.collections4.set.TransformedSortedSet;  
038import org.apache.commons.collections4.set.UnmodifiableNavigableSet;  
039import org.apache.commons.collections4.set.UnmodifiableSet;  
040import org.apache.commons.collections4.set.UnmodifiableSortedSet;  
041  
042/\*\*  
043 \* Provides utility methods and decorators for  
044 \* {@link Set} and {@link SortedSet} instances.  
045 \*  
046 \* @since 2.1  
047 \*/  
048public class SetUtils {  
049  
050 /\*\*  
051 \* An unmodifiable <b>view</b> of a set that may be backed by other sets.  
052 \* <p>  
053 \* If the decorated sets change, this view will change as well. The contents  
054 \* of this view can be transferred to another instance via the {@link #copyInto(Set)}  
055 \* and {@link #toSet()} methods.  
056 \*  
057 \* @param <E> the element type  
058 \* @since 4.1  
059 \*/  
060 public static abstract class SetView<E> extends AbstractSet<E> {  
061  
062 /\*\*  
063 \* Copies the contents of this view into the provided set.  
064 \*  
065 \* @param <S> the set type  
066 \* @param set the set for copying the contents  
067 \*/  
068 public <S extends Set<E>> void copyInto(final S set) {  
069 CollectionUtils.addAll(set, this);  
070 }  
071  
072 /\*\*  
073 \* Return an iterator for this view; the returned iterator is  
074 \* not required to be unmodifiable.  
075 \* @return a new iterator for this view  
076 \*/  
077 protected abstract Iterator<E> createIterator();  
078  
079 @Override  
080 public Iterator<E> iterator() {  
081 return IteratorUtils.unmodifiableIterator(createIterator());  
082 }  
083  
084 @Override  
085 public int size() {  
086 return IteratorUtils.size(iterator());  
087 }  
088  
089 /\*\*  
090 \* Returns a new set containing the contents of this view.  
091 \*  
092 \* @return a new set containing all elements of this view  
093 \*/  
094 public Set<E> toSet() {  
095 final Set<E> set = new HashSet<>(size());  
096 copyInto(set);  
097 return set;  
098 }  
099 }  
100  
101 /\*\*  
102 \* An empty unmodifiable sorted set.  
103 \* This is not provided in the JDK.  
104 \*/  
105 @SuppressWarnings("rawtypes")  
106 public static final SortedSet EMPTY\_SORTED\_SET =  
107 UnmodifiableSortedSet.unmodifiableSortedSet(new TreeSet<>());  
108  
109 /\*\*  
110 \* Returns a unmodifiable <b>view</b> containing the difference of the given  
111 \* {@link Set}s, denoted by {@code a \ b} (or {@code a - b}).  
112 \* <p>  
113 \* The returned view contains all elements of {@code a} that are not a member  
114 \* of {@code b}.  
115 \*  
116 \* @param <E> the generic type that is able to represent the types contained  
117 \* in both input sets.  
118 \* @param a the set to subtract from, must not be null  
119 \* @param b the set to subtract, must not be null  
120 \* @return a view of the relative complement of of the two sets  
121 \* @since 4.1  
122 \*/  
123 public static <E> SetView<E> difference(final Set<? extends E> a, final Set<? extends E> b) {  
124 if (a == null || b == null) {  
125 throw new NullPointerException("Sets must not be null.");  
126 }  
127  
128 final Predicate<E> notContainedInB = new Predicate<E>() {  
129 @Override  
130 public boolean evaluate(final E object) {  
131 return !b.contains(object);  
132 }  
133 };  
134  
135 return new SetView<E>() {  
136 @Override  
137 public boolean contains(final Object o) {  
138 return a.contains(o) && !b.contains(o);  
139 }  
140  
141 @Override  
142 public Iterator<E> createIterator() {  
143 return IteratorUtils.filteredIterator(a.iterator(), notContainedInB);  
144 }  
145 };  
146 }  
147  
148 /\*\*  
149 \* Returns a unmodifiable <b>view</b> of the symmetric difference of the given  
150 \* {@link Set}s.  
151 \* <p>  
152 \* The returned view contains all elements of {@code a} and {@code b} that are  
153 \* not a member of the other set.  
154 \* <p>  
155 \* This is equivalent to {@code union(difference(a, b), difference(b, a))}.  
156 \*  
157 \* @param <E> the generic type that is able to represent the types contained  
158 \* in both input sets.  
159 \* @param a the first set, must not be null  
160 \* @param b the second set, must not be null  
161 \* @return a view of the symmetric difference of the two sets  
162 \* @since 4.1  
163 \*/  
164 public static <E> SetView<E> disjunction(final Set<? extends E> a, final Set<? extends E> b) {  
165 if (a == null || b == null) {  
166 throw new NullPointerException("Sets must not be null.");  
167 }  
168  
169 final SetView<E> aMinusB = difference(a, b);  
170 final SetView<E> bMinusA = difference(b, a);  
171  
172 return new SetView<E>() {  
173 @Override  
174 public boolean contains(final Object o) {  
175 return a.contains(o) ^ b.contains(o);  
176 }  
177  
178 @Override  
179 public Iterator<E> createIterator() {  
180 return IteratorUtils.chainedIterator(aMinusB.iterator(), bMinusA.iterator());  
181 }  
182  
183 @Override  
184 public boolean isEmpty() {  
185 return aMinusB.isEmpty() && bMinusA.isEmpty();  
186 }  
187  
188 @Override  
189 public int size() {  
190 return aMinusB.size() + bMinusA.size();  
191 }  
192 };  
193 }  
194  
195 /\*\*  
196 \* Returns an immutable empty set if the argument is <code>null</code>,  
197 \* or the argument itself otherwise.  
198 \*  
199 \* @param <T> the element type  
200 \* @param set the set, possibly <code>null</code>  
201 \* @return an empty set if the argument is <code>null</code>  
202 \*/  
203 public static <T> Set<T> emptyIfNull(final Set<T> set) {  
204 return set == null ? Collections.<T>emptySet() : set;  
205 }  
206  
207 //-----------------------------------------------------------------------  
208  
209 /\*\*  
210 \* Get a typed empty unmodifiable Set.  
211 \* @param <E> the element type  
212 \* @return an empty Set  
213 \*/  
214 public static <E> Set<E> emptySet() {  
215 return Collections.<E>emptySet();  
216 }  
217  
218 /\*\*  
219 \* Get a typed empty unmodifiable sorted set.  
220 \* @param <E> the element type  
221 \* @return an empty sorted Set  
222 \*/  
223 @SuppressWarnings("unchecked") // empty set is OK for any type  
224 public static <E> SortedSet<E> emptySortedSet() {  
225 return EMPTY\_SORTED\_SET;  
226 }  
227  
228 /\*\*  
229 \* Generates a hash code using the algorithm specified in  
230 \* {@link java.util.Set#hashCode()}.  
231 \* <p>  
232 \* This method is useful for implementing <code>Set</code> when you cannot  
233 \* extend AbstractSet. The method takes Collection instances to enable other  
234 \* collection types to use the Set implementation algorithm.  
235 \*  
236 \* @param <T> the element type  
237 \* @see java.util.Set#hashCode()  
238 \* @param set the set to calculate the hash code for, may be null  
239 \* @return the hash code  
240 \*/  
241 public static <T> int hashCodeForSet(final Collection<T> set) {  
242 if (set == null) {  
243 return 0;  
244 }  
245  
246 int hashCode = 0;  
247 for (final T obj : set) {  
248 if (obj != null) {  
249 hashCode += obj.hashCode();  
250 }  
251 }  
252 return hashCode;  
253 }  
254  
255 /\*\*  
256 \* Creates a set from the given items. If the passed var-args argument is {@code  
257 \* null}, then the method returns {@code null}.  
258 \* @param <E> the element type  
259 \* @param items the elements that make up the new set  
260 \* @return a set  
261 \* @since 4.3  
262 \*/  
263 public static <E> HashSet<E> hashSet(final E... items) {  
264 if (items == null) {  
265 return null;  
266 }  
267 return new HashSet<>(Arrays.asList(items));  
268 }  
269  
270 /\*\*  
271 \* Returns a unmodifiable <b>view</b> of the intersection of the given {@link Set}s.  
272 \* <p>  
273 \* The returned view contains all elements that are members of both input sets  
274 \* ({@code a} and {@code b}).  
275 \*  
276 \* @param <E> the generic type that is able to represent the types contained  
277 \* in both input sets.  
278 \* @param a the first set, must not be null  
279 \* @param b the second set, must not be null  
280 \* @return a view of the intersection of the two sets  
281 \* @since 4.1  
282 \*/  
283 public static <E> SetView<E> intersection(final Set<? extends E> a, final Set<? extends E> b) {  
284 if (a == null || b == null) {  
285 throw new NullPointerException("Sets must not be null.");  
286 }  
287  
288 final Predicate<E> containedInB = new Predicate<E>() {  
289 @Override  
290 public boolean evaluate(final E object) {  
291 return b.contains(object);  
292 }  
293 };  
294  
295 return new SetView<E>() {  
296 @Override  
297 public boolean contains(final Object o) {  
298 return a.contains(o) && b.contains(o);  
299 }  
300  
301 @Override  
302 public Iterator<E> createIterator() {  
303 return IteratorUtils.filteredIterator(a.iterator(), containedInB);  
304 }  
305 };  
306 }  
307  
308 /\*\*  
309 \* Tests two sets for equality as per the <code>equals()</code> contract  
310 \* in {@link java.util.Set#equals(java.lang.Object)}.  
311 \* <p>  
312 \* This method is useful for implementing <code>Set</code> when you cannot  
313 \* extend AbstractSet. The method takes Collection instances to enable other  
314 \* collection types to use the Set implementation algorithm.  
315 \* <p>  
316 \* The relevant text (slightly paraphrased as this is a static method) is:  
317 \* <blockquote>  
318 \* <p>Two sets are considered equal if they have  
319 \* the same size, and every member of the first set is contained in  
320 \* the second. This ensures that the {@code equals} method works  
321 \* properly across different implementations of the {@code Set}  
322 \* interface.</p>  
323 \*  
324 \* <p>  
325 \* This implementation first checks if the two sets are the same object:  
326 \* if so it returns {@code true}. Then, it checks if the two sets are  
327 \* identical in size; if not, it returns false. If so, it returns  
328 \* {@code a.containsAll((Collection) b)}.</p>  
329 \* </blockquote>  
330 \*  
331 \* @see java.util.Set  
332 \* @param set1 the first set, may be null  
333 \* @param set2 the second set, may be null  
334 \* @return whether the sets are equal by value comparison  
335 \*/  
336 public static boolean isEqualSet(final Collection<?> set1, final Collection<?> set2) {  
337 if (set1 == set2) {  
338 return true;  
339 }  
340 if (set1 == null || set2 == null || set1.size() != set2.size()) {  
341 return false;  
342 }  
343  
344 return set1.containsAll(set2);  
345 }  
346  
347 /\*\*  
348 \* Returns a new hash set that matches elements based on <code>==</code> not  
349 \* <code>equals()</code>.  
350 \* <p>  
351 \* <strong>This set will violate the detail of various Set contracts.</strong>  
352 \* As a general rule, don't compare this set to other sets. In particular, you can't  
353 \* use decorators like {@link ListOrderedSet} on it, which silently assume that these  
354 \* contracts are fulfilled.  
355 \* <p>  
356 \* <strong>Note that the returned set is not synchronized and is not thread-safe.</strong>  
357 \* If you wish to use this set from multiple threads concurrently, you must use  
358 \* appropriate synchronization. The simplest approach is to wrap this map  
359 \* using {@link java.util.Collections#synchronizedSet(Set)}. This class may throw  
360 \* exceptions when accessed by concurrent threads without synchronization.  
361 \*  
362 \* @param <E> the element type  
363 \* @return a new identity hash set  
364 \* @since 4.1  
365 \*/  
366 public static <E> Set<E> newIdentityHashSet() {  
367 return Collections.newSetFromMap(new IdentityHashMap<E, Boolean>());  
368 }  
369  
370 /\*\*  
371 \* Returns a set that maintains the order of elements that are added  
372 \* backed by the given set.  
373 \* <p>  
374 \* If an element is added twice, the order is determined by the first add.  
375 \* The order is observed through the iterator or toArray.  
376 \*  
377 \* @param <E> the element type  
378 \* @param set the set to order, must not be null  
379 \* @return an ordered set backed by the given set  
380 \* @throws NullPointerException if the set is null  
381 \*/  
382 public static <E> Set<E> orderedSet(final Set<E> set) {  
383 return ListOrderedSet.listOrderedSet(set);  
384 }  
385  
386 /\*\*  
387 \* Returns a predicated (validating) navigable set backed by the given navigable set.  
388 \* <p>  
389 \* Only objects that pass the test in the given predicate can be added to the set.  
390 \* Trying to add an invalid object results in an IllegalArgumentException.  
391 \* It is important not to use the original set after invoking this method,  
392 \* as it is a backdoor for adding invalid objects.  
393 \*  
394 \* @param <E> the element type  
395 \* @param set the navigable set to predicate, must not be null  
396 \* @param predicate the predicate for the navigable set, must not be null  
397 \* @return a predicated navigable set backed by the given navigable set  
398 \* @throws NullPointerException if the set or predicate is null  
399 \* @since 4.1  
400 \*/  
401 public static <E> SortedSet<E> predicatedNavigableSet(final NavigableSet<E> set,  
402 final Predicate<? super E> predicate) {  
403 return PredicatedNavigableSet.predicatedNavigableSet(set, predicate);  
404 }  
405  
406 /\*\*  
407 \* Returns a predicated (validating) set backed by the given set.  
408 \* <p>  
409 \* Only objects that pass the test in the given predicate can be added to the set.  
410 \* Trying to add an invalid object results in an IllegalArgumentException.  
411 \* It is important not to use the original set after invoking this method,  
412 \* as it is a backdoor for adding invalid objects.  
413 \*  
414 \* @param <E> the element type  
415 \* @param set the set to predicate, must not be null  
416 \* @param predicate the predicate for the set, must not be null  
417 \* @return a predicated set backed by the given set  
418 \* @throws NullPointerException if the set or predicate is null  
419 \*/  
420 public static <E> Set<E> predicatedSet(final Set<E> set, final Predicate<? super E> predicate) {  
421 return PredicatedSet.predicatedSet(set, predicate);  
422 }  
423  
424 /\*\*  
425 \* Returns a predicated (validating) sorted set backed by the given sorted set.  
426 \* <p>  
427 \* Only objects that pass the test in the given predicate can be added to the set.  
428 \* Trying to add an invalid object results in an IllegalArgumentException.  
429 \* It is important not to use the original set after invoking this method,  
430 \* as it is a backdoor for adding invalid objects.  
431 \*  
432 \* @param <E> the element type  
433 \* @param set the sorted set to predicate, must not be null  
434 \* @param predicate the predicate for the sorted set, must not be null  
435 \* @return a predicated sorted set backed by the given sorted set  
436 \* @throws NullPointerException if the set or predicate is null  
437 \*/  
438 public static <E> SortedSet<E> predicatedSortedSet(final SortedSet<E> set,  
439 final Predicate<? super E> predicate) {  
440 return PredicatedSortedSet.predicatedSortedSet(set, predicate);  
441 }  
442  
443 // Set  
444 //-----------------------------------------------------------------------  
445 /\*\*  
446 \* Returns a synchronized set backed by the given set.  
447 \* <p>  
448 \* You must manually synchronize on the returned set's iterator to  
449 \* avoid non-deterministic behavior:  
450 \*  
451 \* <pre>  
452 \* Set s = SetUtils.synchronizedSet(mySet);  
453 \* synchronized (s) {  
454 \* Iterator i = s.iterator();  
455 \* while (i.hasNext()) {  
456 \* process (i.next());  
457 \* }  
458 \* }  
459 \* </pre>  
460 \*  
461 \* This method is just a wrapper for {@link Collections#synchronizedSet(Set)}.  
462 \*  
463 \* @param <E> the element type  
464 \* @param set the set to synchronize, must not be null  
465 \* @return a synchronized set backed by the given set  
466 \* @throws NullPointerException if the set is null  
467 \*/  
468 public static <E> Set<E> synchronizedSet(final Set<E> set) {  
469 return Collections.synchronizedSet(set);  
470 }  
471  
472 // SortedSet  
473 //-----------------------------------------------------------------------  
474 /\*\*  
475 \* Returns a synchronized sorted set backed by the given sorted set.  
476 \* <p>  
477 \* You must manually synchronize on the returned set's iterator to  
478 \* avoid non-deterministic behavior:  
479 \*  
480 \* <pre>  
481 \* Set s = SetUtils.synchronizedSortedSet(mySet);  
482 \* synchronized (s) {  
483 \* Iterator i = s.iterator();  
484 \* while (i.hasNext()) {  
485 \* process (i.next());  
486 \* }  
487 \* }  
488 \* </pre>  
489 \*  
490 \* This method is just a wrapper for {@link Collections#synchronizedSortedSet(SortedSet)}.  
491 \*  
492 \* @param <E> the element type  
493 \* @param set the sorted set to synchronize, must not be null  
494 \* @return a synchronized set backed by the given set  
495 \* @throws NullPointerException if the set is null  
496 \*/  
497 public static <E> SortedSet<E> synchronizedSortedSet(final SortedSet<E> set) {  
498 return Collections.synchronizedSortedSet(set);  
499 }  
500  
501 /\*\*  
502 \* Returns a transformed navigable set backed by the given navigable set.  
503 \* <p>  
504 \* Each object is passed through the transformer as it is added to the  
505 \* Set. It is important not to use the original set after invoking this  
506 \* method, as it is a backdoor for adding untransformed objects.  
507 \* <p>  
508 \* Existing entries in the specified set will not be transformed.  
509 \* If you want that behaviour, see {@link TransformedNavigableSet#transformedNavigableSet}.  
510 \*  
511 \* @param <E> the element type  
512 \* @param set the navigable set to transform, must not be null  
513 \* @param transformer the transformer for the set, must not be null  
514 \* @return a transformed set backed by the given set  
515 \* @throws NullPointerException if the set or transformer is null  
516 \* @since 4.1  
517 \*/  
518 public static <E> SortedSet<E> transformedNavigableSet(final NavigableSet<E> set,  
519 final Transformer<? super E, ? extends E> transformer) {  
520 return TransformedNavigableSet.transformingNavigableSet(set, transformer);  
521 }  
522  
523 /\*\*  
524 \* Returns a transformed set backed by the given set.  
525 \* <p>  
526 \* Each object is passed through the transformer as it is added to the  
527 \* Set. It is important not to use the original set after invoking this  
528 \* method, as it is a backdoor for adding untransformed objects.  
529 \* <p>  
530 \* Existing entries in the specified set will not be transformed.  
531 \* If you want that behaviour, see {@link TransformedSet#transformedSet}.  
532 \*  
533 \* @param <E> the element type  
534 \* @param set the set to transform, must not be null  
535 \* @param transformer the transformer for the set, must not be null  
536 \* @return a transformed set backed by the given set  
537 \* @throws NullPointerException if the set or transformer is null  
538 \*/  
539 public static <E> Set<E> transformedSet(final Set<E> set,  
540 final Transformer<? super E, ? extends E> transformer) {  
541 return TransformedSet.transformingSet(set, transformer);  
542 }  
543  
544 /\*\*  
545 \* Returns a transformed sorted set backed by the given set.  
546 \* <p>  
547 \* Each object is passed through the transformer as it is added to the  
548 \* Set. It is important not to use the original set after invoking this  
549 \* method, as it is a backdoor for adding untransformed objects.  
550 \* <p>  
551 \* Existing entries in the specified set will not be transformed.  
552 \* If you want that behaviour, see {@link TransformedSortedSet#transformedSortedSet}.  
553 \*  
554 \* @param <E> the element type  
555 \* @param set the set to transform, must not be null  
556 \* @param transformer the transformer for the set, must not be null  
557 \* @return a transformed set backed by the given set  
558 \* @throws NullPointerException if the set or transformer is null  
559 \*/  
560 public static <E> SortedSet<E> transformedSortedSet(final SortedSet<E> set,  
561 final Transformer<? super E, ? extends E> transformer) {  
562 return TransformedSortedSet.transformingSortedSet(set, transformer);  
563 }  
564  
565 // Set operations  
566 //-----------------------------------------------------------------------  
567  
568 /\*\*  
569 \* Returns a unmodifiable <b>view</b> of the union of the given {@link Set}s.  
570 \* <p>  
571 \* The returned view contains all elements of {@code a} and {@code b}.  
572 \*  
573 \* @param <E> the generic type that is able to represent the types contained  
574 \* in both input sets.  
575 \* @param a the first set, must not be null  
576 \* @param b the second set, must not be null  
577 \* @return a view of the union of the two set  
578 \* @throws NullPointerException if either input set is null  
579 \* @since 4.1  
580 \*/  
581 public static <E> SetView<E> union(final Set<? extends E> a, final Set<? extends E> b) {  
582 if (a == null || b == null) {  
583 throw new NullPointerException("Sets must not be null.");  
584 }  
585  
586 final SetView<E> bMinusA = difference(b, a);  
587  
588 return new SetView<E>() {  
589 @Override  
590 public boolean contains(final Object o) {  
591 return a.contains(o) || b.contains(o);  
592 }  
593  
594 @Override  
595 public Iterator<E> createIterator() {  
596 return IteratorUtils.chainedIterator(a.iterator(), bMinusA.iterator());  
597 }  
598  
599 @Override  
600 public boolean isEmpty() {  
601 return a.isEmpty() && b.isEmpty();  
602 }  
603  
604 @Override  
605 public int size() {  
606 return a.size() + bMinusA.size();  
607 }  
608 };  
609 }  
610  
611 // NavigableSet  
612 //-----------------------------------------------------------------------  
613 /\*\*  
614 \* Returns an unmodifiable navigable set backed by the given navigable set.  
615 \* <p>  
616 \* This method uses the implementation in the decorators subpackage.  
617 \*  
618 \* @param <E> the element type  
619 \* @param set the navigable set to make unmodifiable, must not be null  
620 \* @return an unmodifiable set backed by the given set  
621 \* @throws NullPointerException if the set is null  
622 \* @since 4.1  
623 \*/  
624 public static <E> SortedSet<E> unmodifiableNavigableSet(final NavigableSet<E> set) {  
625 return UnmodifiableNavigableSet.unmodifiableNavigableSet(set);  
626 }  
627  
628 /\*\*  
629 \* Creates an unmodifiable set from the given items. If the passed var-args argument is {@code  
630 \* null}, then the method returns {@code null}.  
631 \* @param <E> the element type  
632 \* @param items the elements that make up the new set  
633 \* @return a set  
634 \* @since 4.3  
635 \*/  
636 public static <E> Set<E> unmodifiableSet(final E... items) {  
637 if (items == null) {  
638 return null;  
639 }  
640 return UnmodifiableSet.unmodifiableSet(hashSet(items));  
641 }  
642  
643 /\*\*  
644 \* Returns an unmodifiable set backed by the given set.  
645 \* <p>  
646 \* This method uses the implementation in the decorators subpackage.  
647 \*  
648 \* @param <E> the element type  
649 \* @param set the set to make unmodifiable, must not be null  
650 \* @return an unmodifiable set backed by the given set  
651 \* @throws NullPointerException if the set is null  
652 \*/  
653 public static <E> Set<E> unmodifiableSet(final Set<? extends E> set) {  
654 return UnmodifiableSet.unmodifiableSet(set);  
655 }  
656  
657 /\*\*  
658 \* Returns an unmodifiable sorted set backed by the given sorted set.  
659 \* <p>  
660 \* This method uses the implementation in the decorators subpackage.  
661 \*  
662 \* @param <E> the element type  
663 \* @param set the sorted set to make unmodifiable, must not be null  
664 \* @return an unmodifiable set backed by the given set  
665 \* @throws NullPointerException if the set is null  
666 \*/  
667 public static <E> SortedSet<E> unmodifiableSortedSet(final SortedSet<E> set) {  
668 return UnmodifiableSortedSet.unmodifiableSortedSet(set);  
669 }  
670  
671 /\*\*  
672 \* <code>SetUtils</code> should not normally be instantiated.  
673 \*/  
674 private SetUtils() {}  
675}