Оглавление

[Коллекция 3](#_Toc47521954)

[Иерархия 3](#_Toc47521955)

[Почему Map не Collection 3](#_Toc47521956)

[Collection и Collections 4](#_Toc47521957)

[fail-fast и fail-safe поведения 4](#_Toc47521958)

[Enumeration и Iterator 5](#_Toc47521959)

[Iterator и ListIterator 5](#_Toc47521960)

[Iterable, Iterator и «for-each» 6](#_Toc47521961)

[Как поведет себя уже инстанциированный итератор для collection, если вызвать collection.remove()? 6](#_Toc47521962)

[Можно ли итерируясь по ArrayList удалить элемент? Какое исключение? 6](#_Toc47521963)

[Как поведет себя коллекция, если вызвать iterator.remove()? 6](#_Toc47521964)

[List (список) и Set 7](#_Toc47521965)

[Set и его реализации 7](#_Toc47521966)

[TreeSet и HashSet 8](#_Toc47521967)

[Что будет, если добавлять элементы в TreeSet по возрастанию? 9](#_Toc47521968)

[List и его реализации 9](#_Toc47521969)

[ArrayList и LinkedList, сложность основных операций 10](#_Toc47521970)

[Почему LinkedList реализует и List, и Deque? 12](#_Toc47521971)

[Queue 12](#_Toc47521972)

[Устройство HashMap 12](#_Toc47521973)

[Какое худшее время работы метода get(key) для ключа, которого нет в HashMap? 14](#_Toc47521974)

[Какое худшее время работы метода get(key) для ключа, который есть в HashMap? 14](#_Toc47521975)

[Реализации 15](#_Toc47521976)

[Iterable 15](#_Toc47521977)

[Collection 15](#_Toc47521978)

[Collections 18](#_Toc47521979)

[List (список) 185](#_Toc47521980)

[Set 189](#_Toc47521981)

[Queue 190](#_Toc47521982)

# Коллекция

Коллекция — структура данных, набор каких-либо объектов. Данными (объектами в наборе) могут быть числа, строки, объекты пользовательских классов и т. п.

Коллекция — контейнер однотипных элементов.

# Иерархия

Iterable

Collection

Set

List

Queue

ArrayList

Vector

LinkedList

Stack

PriorityQueue

Deque

ArrayDeque

HashSet

LinkedHashSet

TreeSet

SortedSet

Map

Hashtable

LinkedHashMap

WeakHashMap

SortedMap

HashMap

TreeMap

# Почему Map не Collection

Collection представляет собой совокупность некоторых элементов. Map — совокупность пар «ключ-значение».

# Collection и Collections

java.util.Collections — набор статических методов для работы с коллекциями.

java.util.Collection — один из основных интерфейсов Java Collections Framework.

# fail-fast и fail-safe поведения

fail-fast поведение означает, что при возникновении ошибки или состояния, которое может привести к ошибке, система немедленно прекращает дальнейшую работу и уведомляет об этом. Использование fail-fast подхода позволяет избежать недетерминированного поведения программы в течение времени.

В Java Collections API некоторые итераторы ведут себя как fail-fast и выбрасывают ConcurrentModificationException, если после его создания была произведена модификация коллекции, т. е. добавлен или удален элемент напрямую из коллекции, а не используя методы итератора.

Реализация такого поведения осуществляется за счет подсчета количества модификаций коллекции (modification count):  
∙ при изменении коллекции счетчик модификаций так же изменяется;  
∙ при создании итератора ему передается текущее значение счетчика;  
∙ при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.

В противоположность fail-fast, итераторы fail-safe не вызывают никаких исключений при изменении структуры, потому что они работают с клоном коллекции вместо оригинала.

Итератор коллекции CopyOnWriteArrayList и итератор представления keySet коллекции ConcurrentHashMap являются примерами итераторов fail-safe.

# Enumeration и Iterator

Обходят коллекции.

∙ Enumeration присутствует только в устаревших классах Vector и Stack;  
∙ Enumeration не умеет удалять элементы;  
∙ в Iterator упрощена читаемость методов: hasMoreElements() и hasNext(), nextElement() и next().

# Iterator и ListIterator

|  |  |  |
| --- | --- | --- |
| интерфейс | Iterator | ListIterator |
| наследуется от | — | Iterator |
| работает с | Collection | List |
| методы обхода | boolean hasNext()  E next() | |
| — | boolean hasPrevious()  E previous() |
| методы получения индекса | — | int nextIndex()  int previousIndex() |
| методы редактирования коллекций | default void remove() | void remove() |
| — | void set(E e)  void add(E e) |

# Iterable, Iterator и «for-each»

Iterator — тип, возвращаемый iterator() (Iterable).

Классы, реализующие интерфейс Iterable, могут применяться в конструкции for-each, которая использует Iterator.

# Как поведет себя уже инстанциированный итератор для collection, если вызвать collection.remove()?

При следующем вызове методов итератора будет выброшено ConcurrentModificationException.

# Можно ли итерируясь по ArrayList удалить элемент? Какое исключение?

Тут вопрос неккоректный, если используем fail-safe итаротор, то можем удалить, иначе concurrentModificationException

# Как поведет себя коллекция, если вызвать iterator.remove()?

Если вызову iterator.remove() предшествовал вызов iterator.next(), то iterator.remove() удалит элемент коллекции, на который указывает итератор, в  противном случае будет выброшено IllegalStateException().

LinkedList (двунаправленный связный список)

# List (список) и Set

List — пронумерованная (проиндексированная) Collection.

Set — Collection без повторяющихся элементов.

# Set и его реализации

Set

HashSet

LinkedHashSet

TreeSet

SortedSet

Set — описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов.

HashSet — использует HashMap для хранения данных. В качестве ключа и значения используется добавляемый элемент. Из‑за особенностей реализации порядок элементов не гарантируется при добавлении.

LinkedHashSet — гарантирует, что порядок элементов при обходе коллекции будет идентичен порядку добавления элементов.

TreeSet — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».

# TreeSet и HashSet

TreeSet обеспечивает упорядоченно хранение элементов в виде красно-черного дерева. Сложность выполнения основных операций не хуже O(log(N)) (логарифмическое время).

HashSet использует для хранения элементов такой же подход, что и HashMap, за тем отличием, что в HashSet в качестве ключа и значения выступает сам элемент, кроме того HashSet не поддерживает упорядоченное хранение элементов и обеспечивает временную сложность выполнения операций аналогично HashMap.

LinkedHashSet и HashSet

LinkedHashSet отличается от HashSet только тем, что в его основе лежит LinkedHashMap вместо HashMap. Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов (insertion-order). При добавлении элемента, который уже присутствует в LinkedHashSet (т. е. с одинаковым ключом), порядок обхода элементов не изменяется.

# Чем LinkedHashSet отличается от HashSet?

LinkedHashSet отличается от HashSet только тем, что в его основе лежит LinkedHashMap вместо HashMap. Благодаря этому порядок элементов при обходе коллекции является идентичным порядку добавления элементов (insertion-order). При добавлении элемента, который уже присутствует в LinkedHashSet (т. е. с одинаковым ключом), порядок обхода элементов не изменяется.

# Что будет, если добавлять элементы в TreeSet по возрастанию?

В основе TreeSet лежит красно-черное дерево, которое умеет само себя балансировать. В итоге, TreeSet все равно в каком порядке вы добавляете в него элементы, преимущества этой структуры данных будут сохраняться.

# List и его реализации

List

ArrayList

Vector

LinkedList

Stack

List (список) — представляет собой коллекцию, в которой допустимы дублирующие значения. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

ArrayList — инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.

LinkedList (двунаправленный связный список) — состоит из узлов, каждый из которых содержит как собственно данные, так и две ссылки на следующий и предыдущий узел.

Vector — реализация динамического массива объектов, методы которой синхронизированы.

Stack — реализация стека LIFO (last-in-first-out).

# ArrayList и LinkedList, сложность основных операций

ArrayList — список, реализованный на основе массива, а LinkedList — классический двусвязный список, основанный на объектах с ссылками между ними.

ArrayList:  
∙ доступ к произвольному элементу по индексу за константное время O(1);  
∙ доступ к элементам по значению за линейное время O(N);  
∙ вставка в конец в среднем производится за константное время O(1);  
∙ удаление произвольного элемента из списка занимает значительное время, т. к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку влево (реальный размер массива (capacity) не изменяется);  
∙ вставка элемента в произвольное место списка занимает значительное время, т. к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку вправо;  
∙ минимум накладных расходов при хранении.

LinkedList:  
∙ на получение элемента по индексу или значению потребуется линейное время O(N);  
∙ на добавление и удаление в начало или конец списка потребуется константное O(1);  
∙ вставка или удаление в/из произвольного место константное O(1);  
∙ требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка.

В целом, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти, и по скорости выполнения операций. LinkedList предпочтительно применять, когда нужны частые операции вставки/удаления или в случаях, когда необходимо гарантированное время добавления элемента в список.

# Почему LinkedList реализует и List, и Deque?

LinkedList позволяет добавлять элементы в начало и конец списка за константное время, что хорошо согласуется с поведением интерфейса Deque.

# Queue

Queue

LinkedList

PriorityQueue

Deque

ArrayDeque

Queue и Deque

Queue — очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) — соответственно извлечение элемента осуществляется с начала очереди, вставка элемента — в конец очереди. Хотя этот принцип нарушает, к примеру, PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.

Deque (Double Ended Queue) расширяет Queue и согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого, реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.

Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

## Устройство HashMap

HashMap — ассоциативный массив, позволяющий хранить пары «ключ-значение». Каждая ячейка массива — бакет (корзина), хранящая в себе ссылки на списки элементов, узлов (Node).

В корзине может быть один или больше Nod, хранящихся в виде двусвязного списка. При добавлении новой пары «ключ-значение», вычисляется хэш‑код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадает новый элемент.

Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент.

Если в списке был найден элемент с таким же ключом (по equals()), то он заменяется.

Если в списке был найден элемент, ключ которого имеет такой же хэш‑код, но разный equals(), значит произошла коллизия и в списке может быть больше одного узла (Node).

Если в списке все ключи имеют одинаковый хэшкод, но разный equals(), все элементы добавляются в одну корзину и HashMap теряет все свои преимущества, поскольку вырождается в простой двусвязный список элементов.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Временная сложность (среднее/худшее) | | |
| Поиск | Вставка | Удаление |
| TreeSet | O(log(n)) | O(log(n)) | O(log(n)) |
| Hashtable | O(1)  O(n) | O(1)  O(n) | O(1)  O(n) |
| HashMap | O(1)  O(n) | O(1)  O(n) | O(1)  O(n) |
| LinkedHashMap | O(1)  O(n) | O(1)  O(n) | O(1)  O(n) |
| TreeSet | O(log(n)) | O(log(n)) | O(log(n)) |

# Какое худшее время работы метода get(key) для ключа, которого нет в HashMap?

# Какое худшее время работы метода get(key) для ключа, который есть в HashMap?

O(N). Худший случай — поиск ключа в HashMap, вырожденного в список по причине совпадения ключей по hashCode() и для выяснения хранится ли элемент с определенным ключом может потребоваться перебор всего списка.

# Реализации

## **Iterable**

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.lang;  public interface Iterable<T> {  ...  } |

Методы:

|  |  |
| --- | --- |
| 1. | Iterator<T> iterator(); |

вызов Iterator для обхода Collection;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | default void forEach(  Consumer<? super T> action) {  Objects.requireNonNull(action);  for (T t : this) {  action.accept(t);  }  } |

обход Collection при помощи forEach;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | default Spliterator<T> spliterator() {  return Spliterators  .spliteratorUnknownSize(  iterator(), 0);  } |

надо разобраться!!!

## Collection

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package java.util;  public interface Collection<E>  extends Iterable<E> {  ...  } |

Методы:

|  |  |
| --- | --- |
| 1. | int size() |

количество элементов в Collection (не вернет число больше Integer.MAX\_VALUE);

|  |  |
| --- | --- |
| 1. | boolean isEmpty() |

true, если Collection не содержит элементов;

|  |  |
| --- | --- |
| 1. | boolean contains(Object o) |

true, если Collection содержит указанный элемент;

|  |  |
| --- | --- |
| 1. | Iterator<E> iterator() |

вызов Iterator для обхода Collection;

|  |  |
| --- | --- |
| 1. | Object[] toArray() |
| 1. | <T> T[] toArray(T[] a) |

массив из элементов Collection (надо разобрать разницу!!!);

|  |  |
| --- | --- |
| 1. | boolean add(E e) |

добавление элемента;

|  |  |
| --- | --- |
| 1. | boolean remove(Object o) |

удаление элемента;

|  |  |
| --- | --- |
| 1. | boolean containsAll(Collection<?> c); |

true, если Collection содержит все элементы указанной Collection;

|  |  |
| --- | --- |
| 1. | boolean addAll(Collection<? extends E> c) |

добавление элементов указанной Collection;

|  |  |
| --- | --- |
| 1. | boolean removeAll(Collection<?> c); |

удаление элементов указанной Collection;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | default boolean removeIf(  Predicate<? super E> filter) {  Objects.requireNonNull(filter);  boolean removed = false;  final Iterator<E> each = iterator();  while (each.hasNext()) {  if (filter.test(each.next())) {  each.remove();  removed = true;  }  }  return removed;  } |

удаление элементов, удовлетворяющих условию Predicate (boolean test(T t));

|  |  |
| --- | --- |
| 1. | boolean retainAll(Collection<?> c) |

сохранение элементы текущей Collection и переданной Collection;

|  |  |
| --- | --- |
| 1. | void clear() |

удаление всех элементов Collection;

|  |  |
| --- | --- |
| 1. | boolean equals(Object o) |

сравнивает указанный объект с Collection;

|  |  |
| --- | --- |
| 1. | int hashCode() |

хеш‑код Collection;

|  |  |
| --- | --- |
| 1.  2.  3.  4. | @Override  default Spliterator<E> spliterator() {  return Spliterators.spliterator(this, 0);  } |

что такое Spliterator?;

|  |  |
| --- | --- |
| 1.  2.  3.  4. | default Stream<E> stream() {  return StreamSupport  .stream(spliterator(), false);  } |

создание стрима из коллекции;

|  |  |
| --- | --- |
| 1.  2.  3.  4. | default Stream<E> parallelStream() {  return StreamSupport.stream(spliterator(), true);  } |

разобраться с параллельным стримом!!!.

### Collections

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.util;  public class Collections {  ...  } |

Конструкторы:

|  |  |
| --- | --- |
| 1.  2. | private Collections() {  } |

Поля:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | private static final int BINARYSEARCH\_THRESHOLD =  5000;  private static final int REVERSE\_THRESHOLD = 18;  private static final int SHUFFLE\_THRESHOLD = 5;  private static final int FILL\_THRESHOLD = 25;  private static final int ROTATE\_THRESHOLD = 100;  private static final int COPY\_THRESHOLD = 10;  private static final int REPLACEALL\_THRESHOLD =  11;  private static final int  INDEXOFSUBLIST\_THRESHOLD =  35;  private static Random r; |

Методы:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | @SuppressWarnings("unchecked")  public static <T extends Comparable<? super T>>  void sort(List<T> list) {  list.sort(null);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | @SuppressWarnings({"unchecked", "rawtypes"})  public static <T> void sort(  List<T> list,  Comparator<? super T> c) {  list.sort(c);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public static <T> int binarySearch(  List<? extends Comparable<? super T>>  list,  T key) {  if (list instanceof RandomAccess  || list.size()<BINARYSEARCH\_THRESHOLD)  return Collections  .indexedBinarySearch(list, key);  else  return Collections  .iteratorBinarySearch(list, key);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | private static <T> int indexedBinarySearch(  List<? extends Comparable<? super T>>  list,  T key) {  int low = 0;  int high = list.size()-1;  while (low <= high) {  int mid = (low + high) >>> 1;  Comparable<? super T> midVal =  list.get(mid);  int cmp = midVal.compareTo(key);  if (cmp < 0) {  low = mid + 1;  } else if (cmp > 0) {  high = mid - 1;  } else  return mid;  }  }  return -(low + 1);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | private static <T> int iteratorBinarySearch(  List<? extends Comparable<? super T>>  list,  T key) {  int low = 0;  int high = list.size()-1;  ListIterator<? extends Comparable<? super T>>  i =  list.listIterator();  while (low <= high) {  int mid = (low + high) >>> 1;  Comparable<? super T> midVal =  get(i, mid);  int cmp = midVal.compareTo(key);  if (cmp < 0)  low = mid + 1;  else if (cmp > 0)  high = mid - 1;  else  return mid; // key found  }  return -(low + 1); // key not found  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | private static <T> T get(  ListIterator<? extends T> i,  int index) {  T obj = null;  int pos = i.nextIndex();  if (pos <= index) {  do {  obj = i.next();  } while (pos++ < index);  } else {  do {  obj = i.previous();  } while (--pos > index);  }  return obj;  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | @SuppressWarnings("unchecked")  public static <T> int binarySearch(  List<? extends T> list,  T key,  Comparator<? super T> c) {  if (c==null) {  return binarySearch(  (List<? extends Comparable<  ? super T>>) list,  key);  }  if (list instanceof RandomAccess ||  list.size()<BINARYSEARCH\_THRESHOLD) {  return Collections.indexedBinarySearch(  list,  key,  c);  } else {  return Collections.iteratorBinarySearch(  list,  key,  c);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | private static <T> int indexedBinarySearch(  List<? extends T> l,  T key,  Comparator<? super T> c) {  int low = 0;  int high = l.size()-1;  while (low <= high) {  int mid = (low + high) >>> 1;  T midVal = l.get(mid);  int cmp = c.compare(midVal, key);  if (cmp < 0) {  low = mid + 1;  } else if (cmp > 0) {  high = mid - 1;  } else {  return mid;  }  }  return -(low + 1);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | private static <T> int iteratorBinarySearch(  List<? extends T> l,  T key,  Comparator<? super T> c) {  int low = 0;  int high = l.size()-1;  ListIterator<? extends T> i =  l.listIterator();  while (low <= high) {  int mid = (low + high) >>> 1;  T midVal = get(i, mid);  int cmp = c.compare(midVal, key);  if (cmp < 0) {  low = mid + 1;  } else if (cmp > 0) {  high = mid - 1;  } else {  return mid;  }  }  return -(low + 1);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | @SuppressWarnings({"rawtypes", "unchecked"})  public static void reverse(List<?> list) {  int size = list.size();  if (size < REVERSE\_THRESHOLD  || list instanceof RandomAccess) {  for (  int i=0,  mid=size>>1,  j=size-1;  i<mid;  i++,  j--)  swap(list, i, j);  } else {  ListIterator fwd = list.listIterator();  ListIterator rev =  list.listIterator(size);  for (  int i=0,  mid=list.size()>>1;  i<mid;  i++) {  Object tmp = fwd.next();  fwd.set(rev.previous());  rev.set(tmp);  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public static void shuffle(List<?> list) {  Random rnd = r;  if (rnd == null) {  r = rnd = new Random();  }  shuffle(list, rnd);  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | @SuppressWarnings({"rawtypes", "unchecked"})  public static void shuffle(List<?> list, Random rnd) {  int size = list.size();  if (size < SHUFFLE\_THRESHOLD || list instanceof RandomAccess) {  for (int i=size; i>1; i--)  swap(list, i-1, rnd.nextInt(i));  } else {  Object[] arr = list.toArray();  // Shuffle array  for (int i=size; i>1; i--)  swap(arr, i-1, rnd.nextInt(i));  // Dump array back into list  // instead of using a raw type here, it's possible to capture  // the wildcard but it will require a call to a supplementary  // private method  ListIterator it = list.listIterator();  for (int i=0; i<arr.length; i++) {  it.next();  it.set(arr[i]);  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | @SuppressWarnings({"rawtypes", "unchecked"})  public static void shuffle(  List<?> list,  Random rnd) {  int size = list.size();  if (size < SHUFFLE\_THRESHOLD ||  list instanceof RandomAccess) {  for (int i=size; i>1; i--) {  swap(list, i-1, rnd.nextInt(i));  }  } else {  Object[] arr = list.toArray();  for (int i=size; i>1; i--) {  swap(arr, i-1, rnd.nextInt(i));  }  ListIterator it = list.listIterator();  for (int i=0; i<arr.length; i++) {  it.next();  it.set(arr[i]);  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | @SuppressWarnings({"rawtypes", "unchecked"})  public static void swap(  List<?> list,  int i,  int j) {  final List l = list;  l.set(i, l.set(j, l.get(i)));  } |
| 1.  2.  3.  4.  5.  6. | private static void swap(Object[] arr, int i, int j) {  Object tmp = arr[i];  arr[i] = arr[j];  arr[j] = tmp;  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public static <T> void fill(  List<? super T> list,  T obj) {  int size = list.size();  if (size < FILL\_THRESHOLD ||  list instanceof RandomAccess) {  for (int i=0; i<size; i++) {  list.set(i, obj);  }  } else {  ListIterator<? super T> itr =  list.listIterator();  for (int i=0; i<size; i++) {  itr.next();  itr.set(obj);  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | public static <T> void copy(  List<? super T> dest,  List<? extends T> src) {  int srcSize = src.size();  if (srcSize > dest.size()) {  throw new IndexOutOfBoundsException(  "Source does not fit in dest");  }  if (srcSize < COPY\_THRESHOLD  || (src instanceof RandomAccess  && dest instanceof  RandomAccess)) {  for (int i=0; i<srcSize; i++) {  dest.set(i, src.get(i));  }  } else {  ListIterator<? super T> di =  dest.listIterator();  ListIterator<? extends T> si =  src.listIterator();  for (int i=0; i<srcSize; i++) {  di.next();  di.set(si.next());  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. |  |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | /\*\*  \* Returns the minimum element of the given collection, according to the  \* <i>natural ordering</i> of its elements. All elements in the  \* collection must implement the <tt>Comparable</tt> interface.  \* Furthermore, all elements in the collection must be <i>mutually  \* comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a  \* <tt>ClassCastException</tt> for any elements <tt>e1</tt> and  \* <tt>e2</tt> in the collection).<p>  \*  \* This method iterates over the entire collection, hence it requires  \* time proportional to the size of the collection.  \*  \* @param <T> the class of the objects in the collection  \* @param coll the collection whose minimum element is to be determined.  \* @return the minimum element of the given collection, according  \* to the <i>natural ordering</i> of its elements.  \* @throws ClassCastException if the collection contains elements that are  \* not <i>mutually comparable</i> (for example, strings and  \* integers).  \* @throws NoSuchElementException if the collection is empty.  \* @see Comparable  \*/  public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll) {  Iterator<? extends T> i = coll.iterator();  T candidate = i.next();  while (i.hasNext()) {  T next = i.next();  if (next.compareTo(candidate) < 0)  candidate = next;  }  return candidate;  }  /\*\*  \* Returns the minimum element of the given collection, according to the  \* order induced by the specified comparator. All elements in the  \* collection must be <i>mutually comparable</i> by the specified  \* comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a  \* <tt>ClassCastException</tt> for any elements <tt>e1</tt> and  \* <tt>e2</tt> in the collection).<p>  \*  \* This method iterates over the entire collection, hence it requires  \* time proportional to the size of the collection.  \*  \* @param <T> the class of the objects in the collection  \* @param coll the collection whose minimum element is to be determined.  \* @param comp the comparator with which to determine the minimum element.  \* A <tt>null</tt> value indicates that the elements' <i>natural  \* ordering</i> should be used.  \* @return the minimum element of the given collection, according  \* to the specified comparator.  \* @throws ClassCastException if the collection contains elements that are  \* not <i>mutually comparable</i> using the specified comparator.  \* @throws NoSuchElementException if the collection is empty.  \* @see Comparable  \*/  @SuppressWarnings({"unchecked", "rawtypes"})  public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp) {  if (comp==null)  return (T)min((Collection) coll);  Iterator<? extends T> i = coll.iterator();  T candidate = i.next();  while (i.hasNext()) {  T next = i.next();  if (comp.compare(next, candidate) < 0)  candidate = next;  }  return candidate;  }  /\*\*  \* Returns the maximum element of the given collection, according to the  \* <i>natural ordering</i> of its elements. All elements in the  \* collection must implement the <tt>Comparable</tt> interface.  \* Furthermore, all elements in the collection must be <i>mutually  \* comparable</i> (that is, <tt>e1.compareTo(e2)</tt> must not throw a  \* <tt>ClassCastException</tt> for any elements <tt>e1</tt> and  \* <tt>e2</tt> in the collection).<p>  \*  \* This method iterates over the entire collection, hence it requires  \* time proportional to the size of the collection.  \*  \* @param <T> the class of the objects in the collection  \* @param coll the collection whose maximum element is to be determined.  \* @return the maximum element of the given collection, according  \* to the <i>natural ordering</i> of its elements.  \* @throws ClassCastException if the collection contains elements that are  \* not <i>mutually comparable</i> (for example, strings and  \* integers).  \* @throws NoSuchElementException if the collection is empty.  \* @see Comparable  \*/  public static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll) {  Iterator<? extends T> i = coll.iterator();  T candidate = i.next();  while (i.hasNext()) {  T next = i.next();  if (next.compareTo(candidate) > 0)  candidate = next;  }  return candidate;  }  /\*\*  \* Returns the maximum element of the given collection, according to the  \* order induced by the specified comparator. All elements in the  \* collection must be <i>mutually comparable</i> by the specified  \* comparator (that is, <tt>comp.compare(e1, e2)</tt> must not throw a  \* <tt>ClassCastException</tt> for any elements <tt>e1</tt> and  \* <tt>e2</tt> in the collection).<p>  \*  \* This method iterates over the entire collection, hence it requires  \* time proportional to the size of the collection.  \*  \* @param <T> the class of the objects in the collection  \* @param coll the collection whose maximum element is to be determined.  \* @param comp the comparator with which to determine the maximum element.  \* A <tt>null</tt> value indicates that the elements' <i>natural  \* ordering</i> should be used.  \* @return the maximum element of the given collection, according  \* to the specified comparator.  \* @throws ClassCastException if the collection contains elements that are  \* not <i>mutually comparable</i> using the specified comparator.  \* @throws NoSuchElementException if the collection is empty.  \* @see Comparable  \*/  @SuppressWarnings({"unchecked", "rawtypes"})  public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp) {  if (comp==null)  return (T)max((Collection) coll);  Iterator<? extends T> i = coll.iterator();  T candidate = i.next();  while (i.hasNext()) {  T next = i.next();  if (comp.compare(next, candidate) > 0)  candidate = next;  }  return candidate;  }  /\*\*  \* Rotates the elements in the specified list by the specified distance.  \* After calling this method, the element at index <tt>i</tt> will be  \* the element previously at index <tt>(i - distance)</tt> mod  \* <tt>list.size()</tt>, for all values of <tt>i</tt> between <tt>0</tt>  \* and <tt>list.size()-1</tt>, inclusive. (This method has no effect on  \* the size of the list.)  \*  \* <p>For example, suppose <tt>list</tt> comprises<tt> [t, a, n, k, s]</tt>.  \* After invoking <tt>Collections.rotate(list, 1)</tt> (or  \* <tt>Collections.rotate(list, -4)</tt>), <tt>list</tt> will comprise  \* <tt>[s, t, a, n, k]</tt>.  \*  \* <p>Note that this method can usefully be applied to sublists to  \* move one or more elements within a list while preserving the  \* order of the remaining elements. For example, the following idiom  \* moves the element at index <tt>j</tt> forward to position  \* <tt>k</tt> (which must be greater than or equal to <tt>j</tt>):  \* <pre>  \* Collections.rotate(list.subList(j, k+1), -1);  \* </pre>  \* To make this concrete, suppose <tt>list</tt> comprises  \* <tt>[a, b, c, d, e]</tt>. To move the element at index <tt>1</tt>  \* (<tt>b</tt>) forward two positions, perform the following invocation:  \* <pre>  \* Collections.rotate(l.subList(1, 4), -1);  \* </pre>  \* The resulting list is <tt>[a, c, d, b, e]</tt>.  \*  \* <p>To move more than one element forward, increase the absolute value  \* of the rotation distance. To move elements backward, use a positive  \* shift distance.  \*  \* <p>If the specified list is small or implements the {@link  \* RandomAccess} interface, this implementation exchanges the first  \* element into the location it should go, and then repeatedly exchanges  \* the displaced element into the location it should go until a displaced  \* element is swapped into the first element. If necessary, the process  \* is repeated on the second and successive elements, until the rotation  \* is complete. If the specified list is large and doesn't implement the  \* <tt>RandomAccess</tt> interface, this implementation breaks the  \* list into two sublist views around index <tt>-distance mod size</tt>.  \* Then the {@link #reverse(List)} method is invoked on each sublist view,  \* and finally it is invoked on the entire list. For a more complete  \* description of both algorithms, see Section 2.3 of Jon Bentley's  \* <i>Programming Pearls</i> (Addison-Wesley, 1986).  \*  \* @param list the list to be rotated.  \* @param distance the distance to rotate the list. There are no  \* constraints on this value; it may be zero, negative, or  \* greater than <tt>list.size()</tt>.  \* @throws UnsupportedOperationException if the specified list or  \* its list-iterator does not support the <tt>set</tt> operation.  \* @since 1.4  \*/  public static void rotate(List<?> list, int distance) {  if (list instanceof RandomAccess || list.size() < ROTATE\_THRESHOLD)  rotate1(list, distance);  else  rotate2(list, distance);  }  private static <T> void rotate1(List<T> list, int distance) {  int size = list.size();  if (size == 0)  return;  distance = distance % size;  if (distance < 0)  distance += size;  if (distance == 0)  return;  for (int cycleStart = 0, nMoved = 0; nMoved != size; cycleStart++) {  T displaced = list.get(cycleStart);  int i = cycleStart;  do {  i += distance;  if (i >= size)  i -= size;  displaced = list.set(i, displaced);  nMoved ++;  } while (i != cycleStart);  }  }  private static void rotate2(List<?> list, int distance) {  int size = list.size();  if (size == 0)  return;  int mid = -distance % size;  if (mid < 0)  mid += size;  if (mid == 0)  return;  reverse(list.subList(0, mid));  reverse(list.subList(mid, size));  reverse(list);  }  /\*\*  \* Replaces all occurrences of one specified value in a list with another.  \* More formally, replaces with <tt>newVal</tt> each element <tt>e</tt>  \* in <tt>list</tt> such that  \* <tt>(oldVal==null ? e==null : oldVal.equals(e))</tt>.  \* (This method has no effect on the size of the list.)  \*  \* @param <T> the class of the objects in the list  \* @param list the list in which replacement is to occur.  \* @param oldVal the old value to be replaced.  \* @param newVal the new value with which <tt>oldVal</tt> is to be  \* replaced.  \* @return <tt>true</tt> if <tt>list</tt> contained one or more elements  \* <tt>e</tt> such that  \* <tt>(oldVal==null ? e==null : oldVal.equals(e))</tt>.  \* @throws UnsupportedOperationException if the specified list or  \* its list-iterator does not support the <tt>set</tt> operation.  \* @since 1.4  \*/  public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal) {  boolean result = false;  int size = list.size();  if (size < REPLACEALL\_THRESHOLD || list instanceof RandomAccess) {  if (oldVal==null) {  for (int i=0; i<size; i++) {  if (list.get(i)==null) {  list.set(i, newVal);  result = true;  }  }  } else {  for (int i=0; i<size; i++) {  if (oldVal.equals(list.get(i))) {  list.set(i, newVal);  result = true;  }  }  }  } else {  ListIterator<T> itr=list.listIterator();  if (oldVal==null) {  for (int i=0; i<size; i++) {  if (itr.next()==null) {  itr.set(newVal);  result = true;  }  }  } else {  for (int i=0; i<size; i++) {  if (oldVal.equals(itr.next())) {  itr.set(newVal);  result = true;  }  }  }  }  return result;  }  /\*\*  \* Returns the starting position of the first occurrence of the specified  \* target list within the specified source list, or -1 if there is no  \* such occurrence. More formally, returns the lowest index <tt>i</tt>  \* such that {@code source.subList(i, i+target.size()).equals(target)},  \* or -1 if there is no such index. (Returns -1 if  \* {@code target.size() > source.size()})  \*  \* <p>This implementation uses the "brute force" technique of scanning  \* over the source list, looking for a match with the target at each  \* location in turn.  \*  \* @param source the list in which to search for the first occurrence  \* of <tt>target</tt>.  \* @param target the list to search for as a subList of <tt>source</tt>.  \* @return the starting position of the first occurrence of the specified  \* target list within the specified source list, or -1 if there  \* is no such occurrence.  \* @since 1.4  \*/  public static int indexOfSubList(List<?> source, List<?> target) {  int sourceSize = source.size();  int targetSize = target.size();  int maxCandidate = sourceSize - targetSize;  if (sourceSize < INDEXOFSUBLIST\_THRESHOLD ||  (source instanceof RandomAccess&&target instanceof RandomAccess)) {  nextCand:  for (int candidate = 0; candidate <= maxCandidate; candidate++) {  for (int i=0, j=candidate; i<targetSize; i++, j++)  if (!eq(target.get(i), source.get(j)))  continue nextCand; // Element mismatch, try next cand  return candidate; // All elements of candidate matched target  }  } else { // Iterator version of above algorithm  ListIterator<?> si = source.listIterator();  nextCand:  for (int candidate = 0; candidate <= maxCandidate; candidate++) {  ListIterator<?> ti = target.listIterator();  for (int i=0; i<targetSize; i++) {  if (!eq(ti.next(), si.next())) {  // Back up source iterator to next candidate  for (int j=0; j<i; j++)  si.previous();  continue nextCand;  }  }  return candidate;  }  }  return -1; // No candidate matched the target  }  /\*\*  \* Returns the starting position of the last occurrence of the specified  \* target list within the specified source list, or -1 if there is no such  \* occurrence. More formally, returns the highest index <tt>i</tt>  \* such that {@code source.subList(i, i+target.size()).equals(target)},  \* or -1 if there is no such index. (Returns -1 if  \* {@code target.size() > source.size()})  \*  \* <p>This implementation uses the "brute force" technique of iterating  \* over the source list, looking for a match with the target at each  \* location in turn.  \*  \* @param source the list in which to search for the last occurrence  \* of <tt>target</tt>.  \* @param target the list to search for as a subList of <tt>source</tt>.  \* @return the starting position of the last occurrence of the specified  \* target list within the specified source list, or -1 if there  \* is no such occurrence.  \* @since 1.4  \*/  public static int lastIndexOfSubList(List<?> source, List<?> target) {  int sourceSize = source.size();  int targetSize = target.size();  int maxCandidate = sourceSize - targetSize;  if (sourceSize < INDEXOFSUBLIST\_THRESHOLD ||  source instanceof RandomAccess) { // Index access version  nextCand:  for (int candidate = maxCandidate; candidate >= 0; candidate--) {  for (int i=0, j=candidate; i<targetSize; i++, j++)  if (!eq(target.get(i), source.get(j)))  continue nextCand; // Element mismatch, try next cand  return candidate; // All elements of candidate matched target  }  } else { // Iterator version of above algorithm  if (maxCandidate < 0)  return -1;  ListIterator<?> si = source.listIterator(maxCandidate);  nextCand:  for (int candidate = maxCandidate; candidate >= 0; candidate--) {  ListIterator<?> ti = target.listIterator();  for (int i=0; i<targetSize; i++) {  if (!eq(ti.next(), si.next())) {  if (candidate != 0) {  // Back up source iterator to next candidate  for (int j=0; j<=i+1; j++)  si.previous();  }  continue nextCand;  }  }  return candidate;  }  }  return -1; // No candidate matched the target  }  // Unmodifiable Wrappers  /\*\*  \* Returns an unmodifiable view of the specified collection. This method  \* allows modules to provide users with "read-only" access to internal  \* collections. Query operations on the returned collection "read through"  \* to the specified collection, and attempts to modify the returned  \* collection, whether direct or via its iterator, result in an  \* <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned collection does <i>not</i> pass the hashCode and equals  \* operations through to the backing collection, but relies on  \* <tt>Object</tt>'s <tt>equals</tt> and <tt>hashCode</tt> methods. This  \* is necessary to preserve the contracts of these operations in the case  \* that the backing collection is a set or a list.<p>  \*  \* The returned collection will be serializable if the specified collection  \* is serializable.  \*  \* @param <T> the class of the objects in the collection  \* @param c the collection for which an unmodifiable view is to be  \* returned.  \* @return an unmodifiable view of the specified collection.  \*/  public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c) {  return new UnmodifiableCollection<>(c);  }  /\*\*  \* @serial include  \*/  static class UnmodifiableCollection<E> implements Collection<E>, Serializable {  private static final long serialVersionUID = 1820017752578914078L;  final Collection<? extends E> c;  UnmodifiableCollection(Collection<? extends E> c) {  if (c==null)  throw new NullPointerException();  this.c = c;  }  public int size() {return c.size();}  public boolean isEmpty() {return c.isEmpty();}  public boolean contains(Object o) {return c.contains(o);}  public Object[] toArray() {return c.toArray();}  public <T> T[] toArray(T[] a) {return c.toArray(a);}  public String toString() {return c.toString();}  public Iterator<E> iterator() {  return new Iterator<E>() {  private final Iterator<? extends E> i = c.iterator();  public boolean hasNext() {return i.hasNext();}  public E next() {return i.next();}  public void remove() {  throw new UnsupportedOperationException();  }  @Override  public void forEachRemaining(Consumer<? super E> action) {  // Use backing collection version  i.forEachRemaining(action);  }  };  }  public boolean add(E e) {  throw new UnsupportedOperationException();  }  public boolean remove(Object o) {  throw new UnsupportedOperationException();  }  public boolean containsAll(Collection<?> coll) {  return c.containsAll(coll);  }  public boolean addAll(Collection<? extends E> coll) {  throw new UnsupportedOperationException();  }  public boolean removeAll(Collection<?> coll) {  throw new UnsupportedOperationException();  }  public boolean retainAll(Collection<?> coll) {  throw new UnsupportedOperationException();  }  public void clear() {  throw new UnsupportedOperationException();  }  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {  c.forEach(action);  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  throw new UnsupportedOperationException();  }  @SuppressWarnings("unchecked")  @Override  public Spliterator<E> spliterator() {  return (Spliterator<E>)c.spliterator();  }  @SuppressWarnings("unchecked")  @Override  public Stream<E> stream() {  return (Stream<E>)c.stream();  }  @SuppressWarnings("unchecked")  @Override  public Stream<E> parallelStream() {  return (Stream<E>)c.parallelStream();  }  }  /\*\*  \* Returns an unmodifiable view of the specified set. This method allows  \* modules to provide users with "read-only" access to internal sets.  \* Query operations on the returned set "read through" to the specified  \* set, and attempts to modify the returned set, whether direct or via its  \* iterator, result in an <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned set will be serializable if the specified set  \* is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the set for which an unmodifiable view is to be returned.  \* @return an unmodifiable view of the specified set.  \*/  public static <T> Set<T> unmodifiableSet(Set<? extends T> s) {  return new UnmodifiableSet<>(s);  }  /\*\*  \* @serial include  \*/  static class UnmodifiableSet<E> extends UnmodifiableCollection<E>  implements Set<E>, Serializable {  private static final long serialVersionUID = -9215047833775013803L;  UnmodifiableSet(Set<? extends E> s) {super(s);}  public boolean equals(Object o) {return o == this || c.equals(o);}  public int hashCode() {return c.hashCode();}  }  /\*\*  \* Returns an unmodifiable view of the specified sorted set. This method  \* allows modules to provide users with "read-only" access to internal  \* sorted sets. Query operations on the returned sorted set "read  \* through" to the specified sorted set. Attempts to modify the returned  \* sorted set, whether direct, via its iterator, or via its  \* <tt>subSet</tt>, <tt>headSet</tt>, or <tt>tailSet</tt> views, result in  \* an <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned sorted set will be serializable if the specified sorted set  \* is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the sorted set for which an unmodifiable view is to be  \* returned.  \* @return an unmodifiable view of the specified sorted set.  \*/  public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s) {  return new UnmodifiableSortedSet<>(s);  }  /\*\*  \* @serial include  \*/  static class UnmodifiableSortedSet<E>  extends UnmodifiableSet<E>  implements SortedSet<E>, Serializable {  private static final long serialVersionUID = -4929149591599911165L;  private final SortedSet<E> ss;  UnmodifiableSortedSet(SortedSet<E> s) {super(s); ss = s;}  public Comparator<? super E> comparator() {return ss.comparator();}  public SortedSet<E> subSet(E fromElement, E toElement) {  return new UnmodifiableSortedSet<>(ss.subSet(fromElement,toElement));  }  public SortedSet<E> headSet(E toElement) {  return new UnmodifiableSortedSet<>(ss.headSet(toElement));  }  public SortedSet<E> tailSet(E fromElement) {  return new UnmodifiableSortedSet<>(ss.tailSet(fromElement));  }  public E first() {return ss.first();}  public E last() {return ss.last();}  }  /\*\*  \* Returns an unmodifiable view of the specified navigable set. This method  \* allows modules to provide users with "read-only" access to internal  \* navigable sets. Query operations on the returned navigable set "read  \* through" to the specified navigable set. Attempts to modify the returned  \* navigable set, whether direct, via its iterator, or via its  \* {@code subSet}, {@code headSet}, or {@code tailSet} views, result in  \* an {@code UnsupportedOperationException}.<p>  \*  \* The returned navigable set will be serializable if the specified  \* navigable set is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the navigable set for which an unmodifiable view is to be  \* returned  \* @return an unmodifiable view of the specified navigable set  \* @since 1.8  \*/  public static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> s) {  return new UnmodifiableNavigableSet<>(s);  }  /\*\*  \* Wraps a navigable set and disables all of the mutative operations.  \*  \* @param <E> type of elements  \* @serial include  \*/  static class UnmodifiableNavigableSet<E>  extends UnmodifiableSortedSet<E>  implements NavigableSet<E>, Serializable {  private static final long serialVersionUID = -6027448201786391929L;  /\*\*  \* A singleton empty unmodifiable navigable set used for  \* {@link #emptyNavigableSet()}.  \*  \* @param <E> type of elements, if there were any, and bounds  \*/  private static class EmptyNavigableSet<E> extends UnmodifiableNavigableSet<E>  implements Serializable {  private static final long serialVersionUID = -6291252904449939134L;  public EmptyNavigableSet() {  super(new TreeSet<E>());  }  private Object readResolve() { return EMPTY\_NAVIGABLE\_SET; }  }  @SuppressWarnings("rawtypes")  private static final NavigableSet<?> EMPTY\_NAVIGABLE\_SET =  new EmptyNavigableSet<>();  /\*\*  \* The instance we are protecting.  \*/  private final NavigableSet<E> ns;  UnmodifiableNavigableSet(NavigableSet<E> s) {super(s); ns = s;}  public E lower(E e) { return ns.lower(e); }  public E floor(E e) { return ns.floor(e); }  public E ceiling(E e) { return ns.ceiling(e); }  public E higher(E e) { return ns.higher(e); }  public E pollFirst() { throw new UnsupportedOperationException(); }  public E pollLast() { throw new UnsupportedOperationException(); }  public NavigableSet<E> descendingSet()  { return new UnmodifiableNavigableSet<>(ns.descendingSet()); }  public Iterator<E> descendingIterator()  { return descendingSet().iterator(); }  public NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) {  return new UnmodifiableNavigableSet<>(  ns.subSet(fromElement, fromInclusive, toElement, toInclusive));  }  public NavigableSet<E> headSet(E toElement, boolean inclusive) {  return new UnmodifiableNavigableSet<>(  ns.headSet(toElement, inclusive));  }  public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {  return new UnmodifiableNavigableSet<>(  ns.tailSet(fromElement, inclusive));  }  }  /\*\*  \* Returns an unmodifiable view of the specified list. This method allows  \* modules to provide users with "read-only" access to internal  \* lists. Query operations on the returned list "read through" to the  \* specified list, and attempts to modify the returned list, whether  \* direct or via its iterator, result in an  \* <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned list will be serializable if the specified list  \* is serializable. Similarly, the returned list will implement  \* {@link RandomAccess} if the specified list does.  \*  \* @param <T> the class of the objects in the list  \* @param list the list for which an unmodifiable view is to be returned.  \* @return an unmodifiable view of the specified list.  \*/  public static <T> List<T> unmodifiableList(List<? extends T> list) {  return (list instanceof RandomAccess ?  new UnmodifiableRandomAccessList<>(list) :  new UnmodifiableList<>(list));  }  /\*\*  \* @serial include  \*/  static class UnmodifiableList<E> extends UnmodifiableCollection<E>  implements List<E> {  private static final long serialVersionUID = -283967356065247728L;  final List<? extends E> list;  UnmodifiableList(List<? extends E> list) {  super(list);  this.list = list;  }  public boolean equals(Object o) {return o == this || list.equals(o);}  public int hashCode() {return list.hashCode();}  public E get(int index) {return list.get(index);}  public E set(int index, E element) {  throw new UnsupportedOperationException();  }  public void add(int index, E element) {  throw new UnsupportedOperationException();  }  public E remove(int index) {  throw new UnsupportedOperationException();  }  public int indexOf(Object o) {return list.indexOf(o);}  public int lastIndexOf(Object o) {return list.lastIndexOf(o);}  public boolean addAll(int index, Collection<? extends E> c) {  throw new UnsupportedOperationException();  }  @Override  public void replaceAll(UnaryOperator<E> operator) {  throw new UnsupportedOperationException();  }  @Override  public void sort(Comparator<? super E> c) {  throw new UnsupportedOperationException();  }  public ListIterator<E> listIterator() {return listIterator(0);}  public ListIterator<E> listIterator(final int index) {  return new ListIterator<E>() {  private final ListIterator<? extends E> i  = list.listIterator(index);  public boolean hasNext() {return i.hasNext();}  public E next() {return i.next();}  public boolean hasPrevious() {return i.hasPrevious();}  public E previous() {return i.previous();}  public int nextIndex() {return i.nextIndex();}  public int previousIndex() {return i.previousIndex();}  public void remove() {  throw new UnsupportedOperationException();  }  public void set(E e) {  throw new UnsupportedOperationException();  }  public void add(E e) {  throw new UnsupportedOperationException();  }  @Override  public void forEachRemaining(Consumer<? super E> action) {  i.forEachRemaining(action);  }  };  }  public List<E> subList(int fromIndex, int toIndex) {  return new UnmodifiableList<>(list.subList(fromIndex, toIndex));  }  /\*\*  \* UnmodifiableRandomAccessList instances are serialized as  \* UnmodifiableList instances to allow them to be deserialized  \* in pre-1.4 JREs (which do not have UnmodifiableRandomAccessList).  \* This method inverts the transformation. As a beneficial  \* side-effect, it also grafts the RandomAccess marker onto  \* UnmodifiableList instances that were serialized in pre-1.4 JREs.  \*  \* Note: Unfortunately, UnmodifiableRandomAccessList instances  \* serialized in 1.4.1 and deserialized in 1.4 will become  \* UnmodifiableList instances, as this method was missing in 1.4.  \*/  private Object readResolve() {  return (list instanceof RandomAccess  ? new UnmodifiableRandomAccessList<>(list)  : this);  }  }  /\*\*  \* @serial include  \*/  static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>  implements RandomAccess  {  UnmodifiableRandomAccessList(List<? extends E> list) {  super(list);  }  public List<E> subList(int fromIndex, int toIndex) {  return new UnmodifiableRandomAccessList<>(  list.subList(fromIndex, toIndex));  }  private static final long serialVersionUID = -2542308836966382001L;  /\*\*  \* Allows instances to be deserialized in pre-1.4 JREs (which do  \* not have UnmodifiableRandomAccessList). UnmodifiableList has  \* a readResolve method that inverts this transformation upon  \* deserialization.  \*/  private Object writeReplace() {  return new UnmodifiableList<>(list);  }  }  /\*\*  \* Returns an unmodifiable view of the specified map. This method  \* allows modules to provide users with "read-only" access to internal  \* maps. Query operations on the returned map "read through"  \* to the specified map, and attempts to modify the returned  \* map, whether direct or via its collection views, result in an  \* <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned map will be serializable if the specified map  \* is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the map for which an unmodifiable view is to be returned.  \* @return an unmodifiable view of the specified map.  \*/  public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m) {  return new UnmodifiableMap<>(m);  }  /\*\*  \* @serial include  \*/  private static class UnmodifiableMap<K,V> implements Map<K,V>, Serializable {  private static final long serialVersionUID = -1034234728574286014L;  private final Map<? extends K, ? extends V> m;  UnmodifiableMap(Map<? extends K, ? extends V> m) {  if (m==null)  throw new NullPointerException();  this.m = m;  }  public int size() {return m.size();}  public boolean isEmpty() {return m.isEmpty();}  public boolean containsKey(Object key) {return m.containsKey(key);}  public boolean containsValue(Object val) {return m.containsValue(val);}  public V get(Object key) {return m.get(key);}  public V put(K key, V value) {  throw new UnsupportedOperationException();  }  public V remove(Object key) {  throw new UnsupportedOperationException();  }  public void putAll(Map<? extends K, ? extends V> m) {  throw new UnsupportedOperationException();  }  public void clear() {  throw new UnsupportedOperationException();  }  private transient Set<K> keySet;  private transient Set<Map.Entry<K,V>> entrySet;  private transient Collection<V> values;  public Set<K> keySet() {  if (keySet==null)  keySet = unmodifiableSet(m.keySet());  return keySet;  }  public Set<Map.Entry<K,V>> entrySet() {  if (entrySet==null)  entrySet = new UnmodifiableEntrySet<>(m.entrySet());  return entrySet;  }  public Collection<V> values() {  if (values==null)  values = unmodifiableCollection(m.values());  return values;  }  public boolean equals(Object o) {return o == this || m.equals(o);}  public int hashCode() {return m.hashCode();}  public String toString() {return m.toString();}  // Override default methods in Map  @Override  @SuppressWarnings("unchecked")  public V getOrDefault(Object k, V defaultValue) {  // Safe cast as we don't change the value  return ((Map<K, V>)m).getOrDefault(k, defaultValue);  }  @Override  public void forEach(BiConsumer<? super K, ? super V> action) {  m.forEach(action);  }  @Override  public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {  throw new UnsupportedOperationException();  }  @Override  public V putIfAbsent(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public boolean remove(Object key, Object value) {  throw new UnsupportedOperationException();  }  @Override  public boolean replace(K key, V oldValue, V newValue) {  throw new UnsupportedOperationException();  }  @Override  public V replace(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfPresent(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V compute(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V merge(K key, V value,  BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  /\*\*  \* We need this class in addition to UnmodifiableSet as  \* Map.Entries themselves permit modification of the backing Map  \* via their setValue operation. This class is subtle: there are  \* many possible attacks that must be thwarted.  \*  \* @serial include  \*/  static class UnmodifiableEntrySet<K,V>  extends UnmodifiableSet<Map.Entry<K,V>> {  private static final long serialVersionUID = 7854390611657943733L;  @SuppressWarnings({"unchecked", "rawtypes"})  UnmodifiableEntrySet(Set<? extends Map.Entry<? extends K, ? extends V>> s) {  // Need to cast to raw in order to work around a limitation in the type system  super((Set)s);  }  static <K, V> Consumer<Map.Entry<K, V>> entryConsumer(Consumer<? super Entry<K, V>> action) {  return e -> action.accept(new UnmodifiableEntry<>(e));  }  public void forEach(Consumer<? super Entry<K, V>> action) {  Objects.requireNonNull(action);  c.forEach(entryConsumer(action));  }  static final class UnmodifiableEntrySetSpliterator<K, V>  implements Spliterator<Entry<K,V>> {  final Spliterator<Map.Entry<K, V>> s;  UnmodifiableEntrySetSpliterator(Spliterator<Entry<K, V>> s) {  this.s = s;  }  @Override  public boolean tryAdvance(Consumer<? super Entry<K, V>> action) {  Objects.requireNonNull(action);  return s.tryAdvance(entryConsumer(action));  }  @Override  public void forEachRemaining(Consumer<? super Entry<K, V>> action) {  Objects.requireNonNull(action);  s.forEachRemaining(entryConsumer(action));  }  @Override  public Spliterator<Entry<K, V>> trySplit() {  Spliterator<Entry<K, V>> split = s.trySplit();  return split == null  ? null  : new UnmodifiableEntrySetSpliterator<>(split);  }  @Override  public long estimateSize() {  return s.estimateSize();  }  @Override  public long getExactSizeIfKnown() {  return s.getExactSizeIfKnown();  }  @Override  public int characteristics() {  return s.characteristics();  }  @Override  public boolean hasCharacteristics(int characteristics) {  return s.hasCharacteristics(characteristics);  }  @Override  public Comparator<? super Entry<K, V>> getComparator() {  return s.getComparator();  }  }  @SuppressWarnings("unchecked")  public Spliterator<Entry<K,V>> spliterator() {  return new UnmodifiableEntrySetSpliterator<>(  (Spliterator<Map.Entry<K, V>>) c.spliterator());  }  @Override  public Stream<Entry<K,V>> stream() {  return StreamSupport.stream(spliterator(), false);  }  @Override  public Stream<Entry<K,V>> parallelStream() {  return StreamSupport.stream(spliterator(), true);  }  public Iterator<Map.Entry<K,V>> iterator() {  return new Iterator<Map.Entry<K,V>>() {  private final Iterator<? extends Map.Entry<? extends K, ? extends V>> i = c.iterator();  public boolean hasNext() {  return i.hasNext();  }  public Map.Entry<K,V> next() {  return new UnmodifiableEntry<>(i.next());  }  public void remove() {  throw new UnsupportedOperationException();  }  };  }  @SuppressWarnings("unchecked")  public Object[] toArray() {  Object[] a = c.toArray();  for (int i=0; i<a.length; i++)  a[i] = new UnmodifiableEntry<>((Map.Entry<? extends K, ? extends V>)a[i]);  return a;  }  @SuppressWarnings("unchecked")  public <T> T[] toArray(T[] a) {  // We don't pass a to c.toArray, to avoid window of  // vulnerability wherein an unscrupulous multithreaded client  // could get his hands on raw (unwrapped) Entries from c.  Object[] arr = c.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));  for (int i=0; i<arr.length; i++)  arr[i] = new UnmodifiableEntry<>((Map.Entry<? extends K, ? extends V>)arr[i]);  if (arr.length > a.length)  return (T[])arr;  System.arraycopy(arr, 0, a, 0, arr.length);  if (a.length > arr.length)  a[arr.length] = null;  return a;  }  /\*\*  \* This method is overridden to protect the backing set against  \* an object with a nefarious equals function that senses  \* that the equality-candidate is Map.Entry and calls its  \* setValue method.  \*/  public boolean contains(Object o) {  if (!(o instanceof Map.Entry))  return false;  return c.contains(  new UnmodifiableEntry<>((Map.Entry<?,?>) o));  }  /\*\*  \* The next two methods are overridden to protect against  \* an unscrupulous List whose contains(Object o) method senses  \* when o is a Map.Entry, and calls o.setValue.  \*/  public boolean containsAll(Collection<?> coll) {  for (Object e : coll) {  if (!contains(e)) // Invokes safe contains() above  return false;  }  return true;  }  public boolean equals(Object o) {  if (o == this)  return true;  if (!(o instanceof Set))  return false;  Set<?> s = (Set<?>) o;  if (s.size() != c.size())  return false;  return containsAll(s); // Invokes safe containsAll() above  }  /\*\*  \* This "wrapper class" serves two purposes: it prevents  \* the client from modifying the backing Map, by short-circuiting  \* the setValue method, and it protects the backing Map against  \* an ill-behaved Map.Entry that attempts to modify another  \* Map Entry when asked to perform an equality check.  \*/  private static class UnmodifiableEntry<K,V> implements Map.Entry<K,V> {  private Map.Entry<? extends K, ? extends V> e;  UnmodifiableEntry(Map.Entry<? extends K, ? extends V> e)  {this.e = Objects.requireNonNull(e);}  public K getKey() {return e.getKey();}  public V getValue() {return e.getValue();}  public V setValue(V value) {  throw new UnsupportedOperationException();  }  public int hashCode() {return e.hashCode();}  public boolean equals(Object o) {  if (this == o)  return true;  if (!(o instanceof Map.Entry))  return false;  Map.Entry<?,?> t = (Map.Entry<?,?>)o;  return eq(e.getKey(), t.getKey()) &&  eq(e.getValue(), t.getValue());  }  public String toString() {return e.toString();}  }  }  }  /\*\*  \* Returns an unmodifiable view of the specified sorted map. This method  \* allows modules to provide users with "read-only" access to internal  \* sorted maps. Query operations on the returned sorted map "read through"  \* to the specified sorted map. Attempts to modify the returned  \* sorted map, whether direct, via its collection views, or via its  \* <tt>subMap</tt>, <tt>headMap</tt>, or <tt>tailMap</tt> views, result in  \* an <tt>UnsupportedOperationException</tt>.<p>  \*  \* The returned sorted map will be serializable if the specified sorted map  \* is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the sorted map for which an unmodifiable view is to be  \* returned.  \* @return an unmodifiable view of the specified sorted map.  \*/  public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K, ? extends V> m) {  return new UnmodifiableSortedMap<>(m);  }  /\*\*  \* @serial include  \*/  static class UnmodifiableSortedMap<K,V>  extends UnmodifiableMap<K,V>  implements SortedMap<K,V>, Serializable {  private static final long serialVersionUID = -8806743815996713206L;  private final SortedMap<K, ? extends V> sm;  UnmodifiableSortedMap(SortedMap<K, ? extends V> m) {super(m); sm = m; }  public Comparator<? super K> comparator() { return sm.comparator(); }  public SortedMap<K,V> subMap(K fromKey, K toKey)  { return new UnmodifiableSortedMap<>(sm.subMap(fromKey, toKey)); }  public SortedMap<K,V> headMap(K toKey)  { return new UnmodifiableSortedMap<>(sm.headMap(toKey)); }  public SortedMap<K,V> tailMap(K fromKey)  { return new UnmodifiableSortedMap<>(sm.tailMap(fromKey)); }  public K firstKey() { return sm.firstKey(); }  public K lastKey() { return sm.lastKey(); }  }  /\*\*  \* Returns an unmodifiable view of the specified navigable map. This method  \* allows modules to provide users with "read-only" access to internal  \* navigable maps. Query operations on the returned navigable map "read  \* through" to the specified navigable map. Attempts to modify the returned  \* navigable map, whether direct, via its collection views, or via its  \* {@code subMap}, {@code headMap}, or {@code tailMap} views, result in  \* an {@code UnsupportedOperationException}.<p>  \*  \* The returned navigable map will be serializable if the specified  \* navigable map is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the navigable map for which an unmodifiable view is to be  \* returned  \* @return an unmodifiable view of the specified navigable map  \* @since 1.8  \*/  public static <K,V> NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> m) {  return new UnmodifiableNavigableMap<>(m);  }  /\*\*  \* @serial include  \*/  static class UnmodifiableNavigableMap<K,V>  extends UnmodifiableSortedMap<K,V>  implements NavigableMap<K,V>, Serializable {  private static final long serialVersionUID = -4858195264774772197L;  /\*\*  \* A class for the {@link EMPTY\_NAVIGABLE\_MAP} which needs readResolve  \* to preserve singleton property.  \*  \* @param <K> type of keys, if there were any, and of bounds  \* @param <V> type of values, if there were any  \*/  private static class EmptyNavigableMap<K,V> extends UnmodifiableNavigableMap<K,V>  implements Serializable {  private static final long serialVersionUID = -2239321462712562324L;  EmptyNavigableMap() { super(new TreeMap<K,V>()); }  @Override  public NavigableSet<K> navigableKeySet()  { return emptyNavigableSet(); }  private Object readResolve() { return EMPTY\_NAVIGABLE\_MAP; }  }  /\*\*  \* Singleton for {@link emptyNavigableMap()} which is also immutable.  \*/  private static final EmptyNavigableMap<?,?> EMPTY\_NAVIGABLE\_MAP =  new EmptyNavigableMap<>();  /\*\*  \* The instance we wrap and protect.  \*/  private final NavigableMap<K, ? extends V> nm;  UnmodifiableNavigableMap(NavigableMap<K, ? extends V> m)  {super(m); nm = m;}  public K lowerKey(K key) { return nm.lowerKey(key); }  public K floorKey(K key) { return nm.floorKey(key); }  public K ceilingKey(K key) { return nm.ceilingKey(key); }  public K higherKey(K key) { return nm.higherKey(key); }  @SuppressWarnings("unchecked")  public Entry<K, V> lowerEntry(K key) {  Entry<K,V> lower = (Entry<K, V>) nm.lowerEntry(key);  return (null != lower)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(lower)  : null;  }  @SuppressWarnings("unchecked")  public Entry<K, V> floorEntry(K key) {  Entry<K,V> floor = (Entry<K, V>) nm.floorEntry(key);  return (null != floor)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(floor)  : null;  }  @SuppressWarnings("unchecked")  public Entry<K, V> ceilingEntry(K key) {  Entry<K,V> ceiling = (Entry<K, V>) nm.ceilingEntry(key);  return (null != ceiling)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(ceiling)  : null;  }  @SuppressWarnings("unchecked")  public Entry<K, V> higherEntry(K key) {  Entry<K,V> higher = (Entry<K, V>) nm.higherEntry(key);  return (null != higher)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(higher)  : null;  }  @SuppressWarnings("unchecked")  public Entry<K, V> firstEntry() {  Entry<K,V> first = (Entry<K, V>) nm.firstEntry();  return (null != first)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(first)  : null;  }  @SuppressWarnings("unchecked")  public Entry<K, V> lastEntry() {  Entry<K,V> last = (Entry<K, V>) nm.lastEntry();  return (null != last)  ? new UnmodifiableEntrySet.UnmodifiableEntry<>(last)  : null;  }  public Entry<K, V> pollFirstEntry()  { throw new UnsupportedOperationException(); }  public Entry<K, V> pollLastEntry()  { throw new UnsupportedOperationException(); }  public NavigableMap<K, V> descendingMap()  { return unmodifiableNavigableMap(nm.descendingMap()); }  public NavigableSet<K> navigableKeySet()  { return unmodifiableNavigableSet(nm.navigableKeySet()); }  public NavigableSet<K> descendingKeySet()  { return unmodifiableNavigableSet(nm.descendingKeySet()); }  public NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) {  return unmodifiableNavigableMap(  nm.subMap(fromKey, fromInclusive, toKey, toInclusive));  }  public NavigableMap<K, V> headMap(K toKey, boolean inclusive)  { return unmodifiableNavigableMap(nm.headMap(toKey, inclusive)); }  public NavigableMap<K, V> tailMap(K fromKey, boolean inclusive)  { return unmodifiableNavigableMap(nm.tailMap(fromKey, inclusive)); }  }  // Synch Wrappers  /\*\*  \* Returns a synchronized (thread-safe) collection backed by the specified  \* collection. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing collection is accomplished  \* through the returned collection.<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* collection when traversing it via {@link Iterator}, {@link Spliterator}  \* or {@link Stream}:  \* <pre>  \* Collection c = Collections.synchronizedCollection(myCollection);  \* ...  \* synchronized (c) {  \* Iterator i = c.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned collection does <i>not</i> pass the {@code hashCode}  \* and {@code equals} operations through to the backing collection, but  \* relies on {@code Object}'s equals and hashCode methods. This is  \* necessary to preserve the contracts of these operations in the case  \* that the backing collection is a set or a list.<p>  \*  \* The returned collection will be serializable if the specified collection  \* is serializable.  \*  \* @param <T> the class of the objects in the collection  \* @param c the collection to be "wrapped" in a synchronized collection.  \* @return a synchronized view of the specified collection.  \*/  public static <T> Collection<T> synchronizedCollection(Collection<T> c) {  return new SynchronizedCollection<>(c);  }  static <T> Collection<T> synchronizedCollection(Collection<T> c, Object mutex) {  return new SynchronizedCollection<>(c, mutex);  }  /\*\*  \* @serial include  \*/  static class SynchronizedCollection<E> implements Collection<E>, Serializable {  private static final long serialVersionUID = 3053995032091335093L;  final Collection<E> c; // Backing Collection  final Object mutex; // Object on which to synchronize  SynchronizedCollection(Collection<E> c) {  this.c = Objects.requireNonNull(c);  mutex = this;  }  SynchronizedCollection(Collection<E> c, Object mutex) {  this.c = Objects.requireNonNull(c);  this.mutex = Objects.requireNonNull(mutex);  }  public int size() {  synchronized (mutex) {return c.size();}  }  public boolean isEmpty() {  synchronized (mutex) {return c.isEmpty();}  }  public boolean contains(Object o) {  synchronized (mutex) {return c.contains(o);}  }  public Object[] toArray() {  synchronized (mutex) {return c.toArray();}  }  public <T> T[] toArray(T[] a) {  synchronized (mutex) {return c.toArray(a);}  }  public Iterator<E> iterator() {  return c.iterator(); // Must be manually synched by user!  }  public boolean add(E e) {  synchronized (mutex) {return c.add(e);}  }  public boolean remove(Object o) {  synchronized (mutex) {return c.remove(o);}  }  public boolean containsAll(Collection<?> coll) {  synchronized (mutex) {return c.containsAll(coll);}  }  public boolean addAll(Collection<? extends E> coll) {  synchronized (mutex) {return c.addAll(coll);}  }  public boolean removeAll(Collection<?> coll) {  synchronized (mutex) {return c.removeAll(coll);}  }  public boolean retainAll(Collection<?> coll) {  synchronized (mutex) {return c.retainAll(coll);}  }  public void clear() {  synchronized (mutex) {c.clear();}  }  public String toString() {  synchronized (mutex) {return c.toString();}  }  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> consumer) {  synchronized (mutex) {c.forEach(consumer);}  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  synchronized (mutex) {return c.removeIf(filter);}  }  @Override  public Spliterator<E> spliterator() {  return c.spliterator(); // Must be manually synched by user!  }  @Override  public Stream<E> stream() {  return c.stream(); // Must be manually synched by user!  }  @Override  public Stream<E> parallelStream() {  return c.parallelStream(); // Must be manually synched by user!  }  private void writeObject(ObjectOutputStream s) throws IOException {  synchronized (mutex) {s.defaultWriteObject();}  }  }  /\*\*  \* Returns a synchronized (thread-safe) set backed by the specified  \* set. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing set is accomplished  \* through the returned set.<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* set when iterating over it:  \* <pre>  \* Set s = Collections.synchronizedSet(new HashSet());  \* ...  \* synchronized (s) {  \* Iterator i = s.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned set will be serializable if the specified set is  \* serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the set to be "wrapped" in a synchronized set.  \* @return a synchronized view of the specified set.  \*/  public static <T> Set<T> synchronizedSet(Set<T> s) {  return new SynchronizedSet<>(s);  }  static <T> Set<T> synchronizedSet(Set<T> s, Object mutex) {  return new SynchronizedSet<>(s, mutex);  }  /\*\*  \* @serial include  \*/  static class SynchronizedSet<E>  extends SynchronizedCollection<E>  implements Set<E> {  private static final long serialVersionUID = 487447009682186044L;  SynchronizedSet(Set<E> s) {  super(s);  }  SynchronizedSet(Set<E> s, Object mutex) {  super(s, mutex);  }  public boolean equals(Object o) {  if (this == o)  return true;  synchronized (mutex) {return c.equals(o);}  }  public int hashCode() {  synchronized (mutex) {return c.hashCode();}  }  }  /\*\*  \* Returns a synchronized (thread-safe) sorted set backed by the specified  \* sorted set. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing sorted set is accomplished  \* through the returned sorted set (or its views).<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* sorted set when iterating over it or any of its <tt>subSet</tt>,  \* <tt>headSet</tt>, or <tt>tailSet</tt> views.  \* <pre>  \* SortedSet s = Collections.synchronizedSortedSet(new TreeSet());  \* ...  \* synchronized (s) {  \* Iterator i = s.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* or:  \* <pre>  \* SortedSet s = Collections.synchronizedSortedSet(new TreeSet());  \* SortedSet s2 = s.headSet(foo);  \* ...  \* synchronized (s) { // Note: s, not s2!!!  \* Iterator i = s2.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned sorted set will be serializable if the specified  \* sorted set is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the sorted set to be "wrapped" in a synchronized sorted set.  \* @return a synchronized view of the specified sorted set.  \*/  public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s) {  return new SynchronizedSortedSet<>(s);  }  /\*\*  \* @serial include  \*/  static class SynchronizedSortedSet<E>  extends SynchronizedSet<E>  implements SortedSet<E>  {  private static final long serialVersionUID = 8695801310862127406L;  private final SortedSet<E> ss;  SynchronizedSortedSet(SortedSet<E> s) {  super(s);  ss = s;  }  SynchronizedSortedSet(SortedSet<E> s, Object mutex) {  super(s, mutex);  ss = s;  }  public Comparator<? super E> comparator() {  synchronized (mutex) {return ss.comparator();}  }  public SortedSet<E> subSet(E fromElement, E toElement) {  synchronized (mutex) {  return new SynchronizedSortedSet<>(  ss.subSet(fromElement, toElement), mutex);  }  }  public SortedSet<E> headSet(E toElement) {  synchronized (mutex) {  return new SynchronizedSortedSet<>(ss.headSet(toElement), mutex);  }  }  public SortedSet<E> tailSet(E fromElement) {  synchronized (mutex) {  return new SynchronizedSortedSet<>(ss.tailSet(fromElement),mutex);  }  }  public E first() {  synchronized (mutex) {return ss.first();}  }  public E last() {  synchronized (mutex) {return ss.last();}  }  }  /\*\*  \* Returns a synchronized (thread-safe) navigable set backed by the  \* specified navigable set. In order to guarantee serial access, it is  \* critical that <strong>all</strong> access to the backing navigable set is  \* accomplished through the returned navigable set (or its views).<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* navigable set when iterating over it or any of its {@code subSet},  \* {@code headSet}, or {@code tailSet} views.  \* <pre>  \* NavigableSet s = Collections.synchronizedNavigableSet(new TreeSet());  \* ...  \* synchronized (s) {  \* Iterator i = s.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* or:  \* <pre>  \* NavigableSet s = Collections.synchronizedNavigableSet(new TreeSet());  \* NavigableSet s2 = s.headSet(foo, true);  \* ...  \* synchronized (s) { // Note: s, not s2!!!  \* Iterator i = s2.iterator(); // Must be in the synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned navigable set will be serializable if the specified  \* navigable set is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param s the navigable set to be "wrapped" in a synchronized navigable  \* set  \* @return a synchronized view of the specified navigable set  \* @since 1.8  \*/  public static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s) {  return new SynchronizedNavigableSet<>(s);  }  /\*\*  \* @serial include  \*/  static class SynchronizedNavigableSet<E>  extends SynchronizedSortedSet<E>  implements NavigableSet<E>  {  private static final long serialVersionUID = -5505529816273629798L;  private final NavigableSet<E> ns;  SynchronizedNavigableSet(NavigableSet<E> s) {  super(s);  ns = s;  }  SynchronizedNavigableSet(NavigableSet<E> s, Object mutex) {  super(s, mutex);  ns = s;  }  public E lower(E e) { synchronized (mutex) {return ns.lower(e);} }  public E floor(E e) { synchronized (mutex) {return ns.floor(e);} }  public E ceiling(E e) { synchronized (mutex) {return ns.ceiling(e);} }  public E higher(E e) { synchronized (mutex) {return ns.higher(e);} }  public E pollFirst() { synchronized (mutex) {return ns.pollFirst();} }  public E pollLast() { synchronized (mutex) {return ns.pollLast();} }  public NavigableSet<E> descendingSet() {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.descendingSet(), mutex);  }  }  public Iterator<E> descendingIterator()  { synchronized (mutex) { return descendingSet().iterator(); } }  public NavigableSet<E> subSet(E fromElement, E toElement) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.subSet(fromElement, true, toElement, false), mutex);  }  }  public NavigableSet<E> headSet(E toElement) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.headSet(toElement, false), mutex);  }  }  public NavigableSet<E> tailSet(E fromElement) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.tailSet(fromElement, true), mutex);  }  }  public NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.subSet(fromElement, fromInclusive, toElement, toInclusive), mutex);  }  }  public NavigableSet<E> headSet(E toElement, boolean inclusive) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.headSet(toElement, inclusive), mutex);  }  }  public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(ns.tailSet(fromElement, inclusive), mutex);  }  }  }  /\*\*  \* Returns a synchronized (thread-safe) list backed by the specified  \* list. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing list is accomplished  \* through the returned list.<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* list when iterating over it:  \* <pre>  \* List list = Collections.synchronizedList(new ArrayList());  \* ...  \* synchronized (list) {  \* Iterator i = list.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned list will be serializable if the specified list is  \* serializable.  \*  \* @param <T> the class of the objects in the list  \* @param list the list to be "wrapped" in a synchronized list.  \* @return a synchronized view of the specified list.  \*/  public static <T> List<T> synchronizedList(List<T> list) {  return (list instanceof RandomAccess ?  new SynchronizedRandomAccessList<>(list) :  new SynchronizedList<>(list));  }  static <T> List<T> synchronizedList(List<T> list, Object mutex) {  return (list instanceof RandomAccess ?  new SynchronizedRandomAccessList<>(list, mutex) :  new SynchronizedList<>(list, mutex));  }  /\*\*  \* @serial include  \*/  static class SynchronizedList<E>  extends SynchronizedCollection<E>  implements List<E> {  private static final long serialVersionUID = -7754090372962971524L;  final List<E> list;  SynchronizedList(List<E> list) {  super(list);  this.list = list;  }  SynchronizedList(List<E> list, Object mutex) {  super(list, mutex);  this.list = list;  }  public boolean equals(Object o) {  if (this == o)  return true;  synchronized (mutex) {return list.equals(o);}  }  public int hashCode() {  synchronized (mutex) {return list.hashCode();}  }  public E get(int index) {  synchronized (mutex) {return list.get(index);}  }  public E set(int index, E element) {  synchronized (mutex) {return list.set(index, element);}  }  public void add(int index, E element) {  synchronized (mutex) {list.add(index, element);}  }  public E remove(int index) {  synchronized (mutex) {return list.remove(index);}  }  public int indexOf(Object o) {  synchronized (mutex) {return list.indexOf(o);}  }  public int lastIndexOf(Object o) {  synchronized (mutex) {return list.lastIndexOf(o);}  }  public boolean addAll(int index, Collection<? extends E> c) {  synchronized (mutex) {return list.addAll(index, c);}  }  public ListIterator<E> listIterator() {  return list.listIterator(); // Must be manually synched by user  }  public ListIterator<E> listIterator(int index) {  return list.listIterator(index); // Must be manually synched by user  }  public List<E> subList(int fromIndex, int toIndex) {  synchronized (mutex) {  return new SynchronizedList<>(list.subList(fromIndex, toIndex),  mutex);  }  }  @Override  public void replaceAll(UnaryOperator<E> operator) {  synchronized (mutex) {list.replaceAll(operator);}  }  @Override  public void sort(Comparator<? super E> c) {  synchronized (mutex) {list.sort(c);}  }  /\*\*  \* SynchronizedRandomAccessList instances are serialized as  \* SynchronizedList instances to allow them to be deserialized  \* in pre-1.4 JREs (which do not have SynchronizedRandomAccessList).  \* This method inverts the transformation. As a beneficial  \* side-effect, it also grafts the RandomAccess marker onto  \* SynchronizedList instances that were serialized in pre-1.4 JREs.  \*  \* Note: Unfortunately, SynchronizedRandomAccessList instances  \* serialized in 1.4.1 and deserialized in 1.4 will become  \* SynchronizedList instances, as this method was missing in 1.4.  \*/  private Object readResolve() {  return (list instanceof RandomAccess  ? new SynchronizedRandomAccessList<>(list)  : this);  }  }  /\*\*  \* @serial include  \*/  static class SynchronizedRandomAccessList<E>  extends SynchronizedList<E>  implements RandomAccess {  SynchronizedRandomAccessList(List<E> list) {  super(list);  }  SynchronizedRandomAccessList(List<E> list, Object mutex) {  super(list, mutex);  }  public List<E> subList(int fromIndex, int toIndex) {  synchronized (mutex) {  return new SynchronizedRandomAccessList<>(  list.subList(fromIndex, toIndex), mutex);  }  }  private static final long serialVersionUID = 1530674583602358482L;  /\*\*  \* Allows instances to be deserialized in pre-1.4 JREs (which do  \* not have SynchronizedRandomAccessList). SynchronizedList has  \* a readResolve method that inverts this transformation upon  \* deserialization.  \*/  private Object writeReplace() {  return new SynchronizedList<>(list);  }  }  /\*\*  \* Returns a synchronized (thread-safe) map backed by the specified  \* map. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing map is accomplished  \* through the returned map.<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* map when iterating over any of its collection views:  \* <pre>  \* Map m = Collections.synchronizedMap(new HashMap());  \* ...  \* Set s = m.keySet(); // Needn't be in synchronized block  \* ...  \* synchronized (m) { // Synchronizing on m, not s!  \* Iterator i = s.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned map will be serializable if the specified map is  \* serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the map to be "wrapped" in a synchronized map.  \* @return a synchronized view of the specified map.  \*/  public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) {  return new SynchronizedMap<>(m);  }  /\*\*  \* @serial include  \*/  private static class SynchronizedMap<K,V>  implements Map<K,V>, Serializable {  private static final long serialVersionUID = 1978198479659022715L;  private final Map<K,V> m; // Backing Map  final Object mutex; // Object on which to synchronize  SynchronizedMap(Map<K,V> m) {  this.m = Objects.requireNonNull(m);  mutex = this;  }  SynchronizedMap(Map<K,V> m, Object mutex) {  this.m = m;  this.mutex = mutex;  }  public int size() {  synchronized (mutex) {return m.size();}  }  public boolean isEmpty() {  synchronized (mutex) {return m.isEmpty();}  }  public boolean containsKey(Object key) {  synchronized (mutex) {return m.containsKey(key);}  }  public boolean containsValue(Object value) {  synchronized (mutex) {return m.containsValue(value);}  }  public V get(Object key) {  synchronized (mutex) {return m.get(key);}  }  public V put(K key, V value) {  synchronized (mutex) {return m.put(key, value);}  }  public V remove(Object key) {  synchronized (mutex) {return m.remove(key);}  }  public void putAll(Map<? extends K, ? extends V> map) {  synchronized (mutex) {m.putAll(map);}  }  public void clear() {  synchronized (mutex) {m.clear();}  }  private transient Set<K> keySet;  private transient Set<Map.Entry<K,V>> entrySet;  private transient Collection<V> values;  public Set<K> keySet() {  synchronized (mutex) {  if (keySet==null)  keySet = new SynchronizedSet<>(m.keySet(), mutex);  return keySet;  }  }  public Set<Map.Entry<K,V>> entrySet() {  synchronized (mutex) {  if (entrySet==null)  entrySet = new SynchronizedSet<>(m.entrySet(), mutex);  return entrySet;  }  }  public Collection<V> values() {  synchronized (mutex) {  if (values==null)  values = new SynchronizedCollection<>(m.values(), mutex);  return values;  }  }  public boolean equals(Object o) {  if (this == o)  return true;  synchronized (mutex) {return m.equals(o);}  }  public int hashCode() {  synchronized (mutex) {return m.hashCode();}  }  public String toString() {  synchronized (mutex) {return m.toString();}  }  // Override default methods in Map  @Override  public V getOrDefault(Object k, V defaultValue) {  synchronized (mutex) {return m.getOrDefault(k, defaultValue);}  }  @Override  public void forEach(BiConsumer<? super K, ? super V> action) {  synchronized (mutex) {m.forEach(action);}  }  @Override  public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {  synchronized (mutex) {m.replaceAll(function);}  }  @Override  public V putIfAbsent(K key, V value) {  synchronized (mutex) {return m.putIfAbsent(key, value);}  }  @Override  public boolean remove(Object key, Object value) {  synchronized (mutex) {return m.remove(key, value);}  }  @Override  public boolean replace(K key, V oldValue, V newValue) {  synchronized (mutex) {return m.replace(key, oldValue, newValue);}  }  @Override  public V replace(K key, V value) {  synchronized (mutex) {return m.replace(key, value);}  }  @Override  public V computeIfAbsent(K key,  Function<? super K, ? extends V> mappingFunction) {  synchronized (mutex) {return m.computeIfAbsent(key, mappingFunction);}  }  @Override  public V computeIfPresent(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  synchronized (mutex) {return m.computeIfPresent(key, remappingFunction);}  }  @Override  public V compute(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  synchronized (mutex) {return m.compute(key, remappingFunction);}  }  @Override  public V merge(K key, V value,  BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  synchronized (mutex) {return m.merge(key, value, remappingFunction);}  }  private void writeObject(ObjectOutputStream s) throws IOException {  synchronized (mutex) {s.defaultWriteObject();}  }  }  /\*\*  \* Returns a synchronized (thread-safe) sorted map backed by the specified  \* sorted map. In order to guarantee serial access, it is critical that  \* <strong>all</strong> access to the backing sorted map is accomplished  \* through the returned sorted map (or its views).<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* sorted map when iterating over any of its collection views, or the  \* collections views of any of its <tt>subMap</tt>, <tt>headMap</tt> or  \* <tt>tailMap</tt> views.  \* <pre>  \* SortedMap m = Collections.synchronizedSortedMap(new TreeMap());  \* ...  \* Set s = m.keySet(); // Needn't be in synchronized block  \* ...  \* synchronized (m) { // Synchronizing on m, not s!  \* Iterator i = s.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* or:  \* <pre>  \* SortedMap m = Collections.synchronizedSortedMap(new TreeMap());  \* SortedMap m2 = m.subMap(foo, bar);  \* ...  \* Set s2 = m2.keySet(); // Needn't be in synchronized block  \* ...  \* synchronized (m) { // Synchronizing on m, not m2 or s2!  \* Iterator i = s.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned sorted map will be serializable if the specified  \* sorted map is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the sorted map to be "wrapped" in a synchronized sorted map.  \* @return a synchronized view of the specified sorted map.  \*/  public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m) {  return new SynchronizedSortedMap<>(m);  }  /\*\*  \* @serial include  \*/  static class SynchronizedSortedMap<K,V>  extends SynchronizedMap<K,V>  implements SortedMap<K,V>  {  private static final long serialVersionUID = -8798146769416483793L;  private final SortedMap<K,V> sm;  SynchronizedSortedMap(SortedMap<K,V> m) {  super(m);  sm = m;  }  SynchronizedSortedMap(SortedMap<K,V> m, Object mutex) {  super(m, mutex);  sm = m;  }  public Comparator<? super K> comparator() {  synchronized (mutex) {return sm.comparator();}  }  public SortedMap<K,V> subMap(K fromKey, K toKey) {  synchronized (mutex) {  return new SynchronizedSortedMap<>(  sm.subMap(fromKey, toKey), mutex);  }  }  public SortedMap<K,V> headMap(K toKey) {  synchronized (mutex) {  return new SynchronizedSortedMap<>(sm.headMap(toKey), mutex);  }  }  public SortedMap<K,V> tailMap(K fromKey) {  synchronized (mutex) {  return new SynchronizedSortedMap<>(sm.tailMap(fromKey),mutex);  }  }  public K firstKey() {  synchronized (mutex) {return sm.firstKey();}  }  public K lastKey() {  synchronized (mutex) {return sm.lastKey();}  }  }  /\*\*  \* Returns a synchronized (thread-safe) navigable map backed by the  \* specified navigable map. In order to guarantee serial access, it is  \* critical that <strong>all</strong> access to the backing navigable map is  \* accomplished through the returned navigable map (or its views).<p>  \*  \* It is imperative that the user manually synchronize on the returned  \* navigable map when iterating over any of its collection views, or the  \* collections views of any of its {@code subMap}, {@code headMap} or  \* {@code tailMap} views.  \* <pre>  \* NavigableMap m = Collections.synchronizedNavigableMap(new TreeMap());  \* ...  \* Set s = m.keySet(); // Needn't be in synchronized block  \* ...  \* synchronized (m) { // Synchronizing on m, not s!  \* Iterator i = s.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* or:  \* <pre>  \* NavigableMap m = Collections.synchronizedNavigableMap(new TreeMap());  \* NavigableMap m2 = m.subMap(foo, true, bar, false);  \* ...  \* Set s2 = m2.keySet(); // Needn't be in synchronized block  \* ...  \* synchronized (m) { // Synchronizing on m, not m2 or s2!  \* Iterator i = s.iterator(); // Must be in synchronized block  \* while (i.hasNext())  \* foo(i.next());  \* }  \* </pre>  \* Failure to follow this advice may result in non-deterministic behavior.  \*  \* <p>The returned navigable map will be serializable if the specified  \* navigable map is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the navigable map to be "wrapped" in a synchronized navigable  \* map  \* @return a synchronized view of the specified navigable map.  \* @since 1.8  \*/  public static <K,V> NavigableMap<K,V> synchronizedNavigableMap(NavigableMap<K,V> m) {  return new SynchronizedNavigableMap<>(m);  }  /\*\*  \* A synchronized NavigableMap.  \*  \* @serial include  \*/  static class SynchronizedNavigableMap<K,V>  extends SynchronizedSortedMap<K,V>  implements NavigableMap<K,V>  {  private static final long serialVersionUID = 699392247599746807L;  private final NavigableMap<K,V> nm;  SynchronizedNavigableMap(NavigableMap<K,V> m) {  super(m);  nm = m;  }  SynchronizedNavigableMap(NavigableMap<K,V> m, Object mutex) {  super(m, mutex);  nm = m;  }  public Entry<K, V> lowerEntry(K key)  { synchronized (mutex) { return nm.lowerEntry(key); } }  public K lowerKey(K key)  { synchronized (mutex) { return nm.lowerKey(key); } }  public Entry<K, V> floorEntry(K key)  { synchronized (mutex) { return nm.floorEntry(key); } }  public K floorKey(K key)  { synchronized (mutex) { return nm.floorKey(key); } }  public Entry<K, V> ceilingEntry(K key)  { synchronized (mutex) { return nm.ceilingEntry(key); } }  public K ceilingKey(K key)  { synchronized (mutex) { return nm.ceilingKey(key); } }  public Entry<K, V> higherEntry(K key)  { synchronized (mutex) { return nm.higherEntry(key); } }  public K higherKey(K key)  { synchronized (mutex) { return nm.higherKey(key); } }  public Entry<K, V> firstEntry()  { synchronized (mutex) { return nm.firstEntry(); } }  public Entry<K, V> lastEntry()  { synchronized (mutex) { return nm.lastEntry(); } }  public Entry<K, V> pollFirstEntry()  { synchronized (mutex) { return nm.pollFirstEntry(); } }  public Entry<K, V> pollLastEntry()  { synchronized (mutex) { return nm.pollLastEntry(); } }  public NavigableMap<K, V> descendingMap() {  synchronized (mutex) {  return  new SynchronizedNavigableMap<>(nm.descendingMap(), mutex);  }  }  public NavigableSet<K> keySet() {  return navigableKeySet();  }  public NavigableSet<K> navigableKeySet() {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(nm.navigableKeySet(), mutex);  }  }  public NavigableSet<K> descendingKeySet() {  synchronized (mutex) {  return new SynchronizedNavigableSet<>(nm.descendingKeySet(), mutex);  }  }  public SortedMap<K,V> subMap(K fromKey, K toKey) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(  nm.subMap(fromKey, true, toKey, false), mutex);  }  }  public SortedMap<K,V> headMap(K toKey) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(nm.headMap(toKey, false), mutex);  }  }  public SortedMap<K,V> tailMap(K fromKey) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(nm.tailMap(fromKey, true),mutex);  }  }  public NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(  nm.subMap(fromKey, fromInclusive, toKey, toInclusive), mutex);  }  }  public NavigableMap<K, V> headMap(K toKey, boolean inclusive) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(  nm.headMap(toKey, inclusive), mutex);  }  }  public NavigableMap<K, V> tailMap(K fromKey, boolean inclusive) {  synchronized (mutex) {  return new SynchronizedNavigableMap<>(  nm.tailMap(fromKey, inclusive), mutex);  }  }  }  // Dynamically typesafe collection wrappers  /\*\*  \* Returns a dynamically typesafe view of the specified collection.  \* Any attempt to insert an element of the wrong type will result in an  \* immediate {@link ClassCastException}. Assuming a collection  \* contains no incorrectly typed elements prior to the time a  \* dynamically typesafe view is generated, and that all subsequent  \* access to the collection takes place through the view, it is  \* <i>guaranteed</i> that the collection cannot contain an incorrectly  \* typed element.  \*  \* <p>The generics mechanism in the language provides compile-time  \* (static) type checking, but it is possible to defeat this mechanism  \* with unchecked casts. Usually this is not a problem, as the compiler  \* issues warnings on all such unchecked operations. There are, however,  \* times when static type checking alone is not sufficient. For example,  \* suppose a collection is passed to a third-party library and it is  \* imperative that the library code not corrupt the collection by  \* inserting an element of the wrong type.  \*  \* <p>Another use of dynamically typesafe views is debugging. Suppose a  \* program fails with a {@code ClassCastException}, indicating that an  \* incorrectly typed element was put into a parameterized collection.  \* Unfortunately, the exception can occur at any time after the erroneous  \* element is inserted, so it typically provides little or no information  \* as to the real source of the problem. If the problem is reproducible,  \* one can quickly determine its source by temporarily modifying the  \* program to wrap the collection with a dynamically typesafe view.  \* For example, this declaration:  \* <pre> {@code  \* Collection<String> c = new HashSet<>();  \* }</pre>  \* may be replaced temporarily by this one:  \* <pre> {@code  \* Collection<String> c = Collections.checkedCollection(  \* new HashSet<>(), String.class);  \* }</pre>  \* Running the program again will cause it to fail at the point where  \* an incorrectly typed element is inserted into the collection, clearly  \* identifying the source of the problem. Once the problem is fixed, the  \* modified declaration may be reverted back to the original.  \*  \* <p>The returned collection does <i>not</i> pass the hashCode and equals  \* operations through to the backing collection, but relies on  \* {@code Object}'s {@code equals} and {@code hashCode} methods. This  \* is necessary to preserve the contracts of these operations in the case  \* that the backing collection is a set or a list.  \*  \* <p>The returned collection will be serializable if the specified  \* collection is serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned collection permits insertion of null elements  \* whenever the backing collection does.  \*  \* @param <E> the class of the objects in the collection  \* @param c the collection for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code c} is permitted to hold  \* @return a dynamically typesafe view of the specified collection  \* @since 1.5  \*/  public static <E> Collection<E> checkedCollection(Collection<E> c,  Class<E> type) {  return new CheckedCollection<>(c, type);  }  @SuppressWarnings("unchecked")  static <T> T[] zeroLengthArray(Class<T> type) {  return (T[]) Array.newInstance(type, 0);  }  /\*\*  \* @serial include  \*/  static class CheckedCollection<E> implements Collection<E>, Serializable {  private static final long serialVersionUID = 1578914078182001775L;  final Collection<E> c;  final Class<E> type;  @SuppressWarnings("unchecked")  E typeCheck(Object o) {  if (o != null && !type.isInstance(o))  throw new ClassCastException(badElementMsg(o));  return (E) o;  }  private String badElementMsg(Object o) {  return "Attempt to insert " + o.getClass() +  " element into collection with element type " + type;  }  CheckedCollection(Collection<E> c, Class<E> type) {  this.c = Objects.requireNonNull(c, "c");  this.type = Objects.requireNonNull(type, "type");  }  public int size() { return c.size(); }  public boolean isEmpty() { return c.isEmpty(); }  public boolean contains(Object o) { return c.contains(o); }  public Object[] toArray() { return c.toArray(); }  public <T> T[] toArray(T[] a) { return c.toArray(a); }  public String toString() { return c.toString(); }  public boolean remove(Object o) { return c.remove(o); }  public void clear() { c.clear(); }  public boolean containsAll(Collection<?> coll) {  return c.containsAll(coll);  }  public boolean removeAll(Collection<?> coll) {  return c.removeAll(coll);  }  public boolean retainAll(Collection<?> coll) {  return c.retainAll(coll);  }  public Iterator<E> iterator() {  // JDK-6363904 - unwrapped iterator could be typecast to  // ListIterator with unsafe set()  final Iterator<E> it = c.iterator();  return new Iterator<E>() {  public boolean hasNext() { return it.hasNext(); }  public E next() { return it.next(); }  public void remove() { it.remove(); }};  }  public boolean add(E e) { return c.add(typeCheck(e)); }  private E[] zeroLengthElementArray; // Lazily initialized  private E[] zeroLengthElementArray() {  return zeroLengthElementArray != null ? zeroLengthElementArray :  (zeroLengthElementArray = zeroLengthArray(type));  }  @SuppressWarnings("unchecked")  Collection<E> checkedCopyOf(Collection<? extends E> coll) {  Object[] a;  try {  E[] z = zeroLengthElementArray();  a = coll.toArray(z);  // Defend against coll violating the toArray contract  if (a.getClass() != z.getClass())  a = Arrays.copyOf(a, a.length, z.getClass());  } catch (ArrayStoreException ignore) {  // To get better and consistent diagnostics,  // we call typeCheck explicitly on each element.  // We call clone() to defend against coll retaining a  // reference to the returned array and storing a bad  // element into it after it has been type checked.  a = coll.toArray().clone();  for (Object o : a)  typeCheck(o);  }  // A slight abuse of the type system, but safe here.  return (Collection<E>) Arrays.asList(a);  }  public boolean addAll(Collection<? extends E> coll) {  // Doing things this way insulates us from concurrent changes  // in the contents of coll and provides all-or-nothing  // semantics (which we wouldn't get if we type-checked each  // element as we added it)  return c.addAll(checkedCopyOf(coll));  }  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {c.forEach(action);}  @Override  public boolean removeIf(Predicate<? super E> filter) {  return c.removeIf(filter);  }  @Override  public Spliterator<E> spliterator() {return c.spliterator();}  @Override  public Stream<E> stream() {return c.stream();}  @Override  public Stream<E> parallelStream() {return c.parallelStream();}  }  /\*\*  \* Returns a dynamically typesafe view of the specified queue.  \* Any attempt to insert an element of the wrong type will result in  \* an immediate {@link ClassCastException}. Assuming a queue contains  \* no incorrectly typed elements prior to the time a dynamically typesafe  \* view is generated, and that all subsequent access to the queue  \* takes place through the view, it is <i>guaranteed</i> that the  \* queue cannot contain an incorrectly typed element.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned queue will be serializable if the specified queue  \* is serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned queue permits insertion of {@code null} elements  \* whenever the backing queue does.  \*  \* @param <E> the class of the objects in the queue  \* @param queue the queue for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code queue} is permitted to hold  \* @return a dynamically typesafe view of the specified queue  \* @since 1.8  \*/  public static <E> Queue<E> checkedQueue(Queue<E> queue, Class<E> type) {  return new CheckedQueue<>(queue, type);  }  /\*\*  \* @serial include  \*/  static class CheckedQueue<E>  extends CheckedCollection<E>  implements Queue<E>, Serializable  {  private static final long serialVersionUID = 1433151992604707767L;  final Queue<E> queue;  CheckedQueue(Queue<E> queue, Class<E> elementType) {  super(queue, elementType);  this.queue = queue;  }  public E element() {return queue.element();}  public boolean equals(Object o) {return o == this || c.equals(o);}  public int hashCode() {return c.hashCode();}  public E peek() {return queue.peek();}  public E poll() {return queue.poll();}  public E remove() {return queue.remove();}  public boolean offer(E e) {return queue.offer(typeCheck(e));}  }  /\*\*  \* Returns a dynamically typesafe view of the specified set.  \* Any attempt to insert an element of the wrong type will result in  \* an immediate {@link ClassCastException}. Assuming a set contains  \* no incorrectly typed elements prior to the time a dynamically typesafe  \* view is generated, and that all subsequent access to the set  \* takes place through the view, it is <i>guaranteed</i> that the  \* set cannot contain an incorrectly typed element.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned set will be serializable if the specified set is  \* serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned set permits insertion of null elements whenever  \* the backing set does.  \*  \* @param <E> the class of the objects in the set  \* @param s the set for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code s} is permitted to hold  \* @return a dynamically typesafe view of the specified set  \* @since 1.5  \*/  public static <E> Set<E> checkedSet(Set<E> s, Class<E> type) {  return new CheckedSet<>(s, type);  }  /\*\*  \* @serial include  \*/  static class CheckedSet<E> extends CheckedCollection<E>  implements Set<E>, Serializable  {  private static final long serialVersionUID = 4694047833775013803L;  CheckedSet(Set<E> s, Class<E> elementType) { super(s, elementType); }  public boolean equals(Object o) { return o == this || c.equals(o); }  public int hashCode() { return c.hashCode(); }  }  /\*\*  \* Returns a dynamically typesafe view of the specified sorted set.  \* Any attempt to insert an element of the wrong type will result in an  \* immediate {@link ClassCastException}. Assuming a sorted set  \* contains no incorrectly typed elements prior to the time a  \* dynamically typesafe view is generated, and that all subsequent  \* access to the sorted set takes place through the view, it is  \* <i>guaranteed</i> that the sorted set cannot contain an incorrectly  \* typed element.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned sorted set will be serializable if the specified sorted  \* set is serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned sorted set permits insertion of null elements  \* whenever the backing sorted set does.  \*  \* @param <E> the class of the objects in the set  \* @param s the sorted set for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code s} is permitted to hold  \* @return a dynamically typesafe view of the specified sorted set  \* @since 1.5  \*/  public static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s,  Class<E> type) {  return new CheckedSortedSet<>(s, type);  }  /\*\*  \* @serial include  \*/  static class CheckedSortedSet<E> extends CheckedSet<E>  implements SortedSet<E>, Serializable  {  private static final long serialVersionUID = 1599911165492914959L;  private final SortedSet<E> ss;  CheckedSortedSet(SortedSet<E> s, Class<E> type) {  super(s, type);  ss = s;  }  public Comparator<? super E> comparator() { return ss.comparator(); }  public E first() { return ss.first(); }  public E last() { return ss.last(); }  public SortedSet<E> subSet(E fromElement, E toElement) {  return checkedSortedSet(ss.subSet(fromElement,toElement), type);  }  public SortedSet<E> headSet(E toElement) {  return checkedSortedSet(ss.headSet(toElement), type);  }  public SortedSet<E> tailSet(E fromElement) {  return checkedSortedSet(ss.tailSet(fromElement), type);  }  }  /\*\*  \* Returns a dynamically typesafe view of the specified navigable set.  \* Any attempt to insert an element of the wrong type will result in an  \* immediate {@link ClassCastException}. Assuming a navigable set  \* contains no incorrectly typed elements prior to the time a  \* dynamically typesafe view is generated, and that all subsequent  \* access to the navigable set takes place through the view, it is  \* <em>guaranteed</em> that the navigable set cannot contain an incorrectly  \* typed element.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned navigable set will be serializable if the specified  \* navigable set is serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned navigable set permits insertion of null elements  \* whenever the backing sorted set does.  \*  \* @param <E> the class of the objects in the set  \* @param s the navigable set for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code s} is permitted to hold  \* @return a dynamically typesafe view of the specified navigable set  \* @since 1.8  \*/  public static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> s,  Class<E> type) {  return new CheckedNavigableSet<>(s, type);  }  /\*\*  \* @serial include  \*/  static class CheckedNavigableSet<E> extends CheckedSortedSet<E>  implements NavigableSet<E>, Serializable  {  private static final long serialVersionUID = -5429120189805438922L;  private final NavigableSet<E> ns;  CheckedNavigableSet(NavigableSet<E> s, Class<E> type) {  super(s, type);  ns = s;  }  public E lower(E e) { return ns.lower(e); }  public E floor(E e) { return ns.floor(e); }  public E ceiling(E e) { return ns.ceiling(e); }  public E higher(E e) { return ns.higher(e); }  public E pollFirst() { return ns.pollFirst(); }  public E pollLast() {return ns.pollLast(); }  public NavigableSet<E> descendingSet()  { return checkedNavigableSet(ns.descendingSet(), type); }  public Iterator<E> descendingIterator()  {return checkedNavigableSet(ns.descendingSet(), type).iterator(); }  public NavigableSet<E> subSet(E fromElement, E toElement) {  return checkedNavigableSet(ns.subSet(fromElement, true, toElement, false), type);  }  public NavigableSet<E> headSet(E toElement) {  return checkedNavigableSet(ns.headSet(toElement, false), type);  }  public NavigableSet<E> tailSet(E fromElement) {  return checkedNavigableSet(ns.tailSet(fromElement, true), type);  }  public NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) {  return checkedNavigableSet(ns.subSet(fromElement, fromInclusive, toElement, toInclusive), type);  }  public NavigableSet<E> headSet(E toElement, boolean inclusive) {  return checkedNavigableSet(ns.headSet(toElement, inclusive), type);  }  public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {  return checkedNavigableSet(ns.tailSet(fromElement, inclusive), type);  }  }  /\*\*  \* Returns a dynamically typesafe view of the specified list.  \* Any attempt to insert an element of the wrong type will result in  \* an immediate {@link ClassCastException}. Assuming a list contains  \* no incorrectly typed elements prior to the time a dynamically typesafe  \* view is generated, and that all subsequent access to the list  \* takes place through the view, it is <i>guaranteed</i> that the  \* list cannot contain an incorrectly typed element.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned list will be serializable if the specified list  \* is serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned list permits insertion of null elements whenever  \* the backing list does.  \*  \* @param <E> the class of the objects in the list  \* @param list the list for which a dynamically typesafe view is to be  \* returned  \* @param type the type of element that {@code list} is permitted to hold  \* @return a dynamically typesafe view of the specified list  \* @since 1.5  \*/  public static <E> List<E> checkedList(List<E> list, Class<E> type) {  return (list instanceof RandomAccess ?  new CheckedRandomAccessList<>(list, type) :  new CheckedList<>(list, type));  }  /\*\*  \* @serial include  \*/  static class CheckedList<E>  extends CheckedCollection<E>  implements List<E>  {  private static final long serialVersionUID = 65247728283967356L;  final List<E> list;  CheckedList(List<E> list, Class<E> type) {  super(list, type);  this.list = list;  }  public boolean equals(Object o) { return o == this || list.equals(o); }  public int hashCode() { return list.hashCode(); }  public E get(int index) { return list.get(index); }  public E remove(int index) { return list.remove(index); }  public int indexOf(Object o) { return list.indexOf(o); }  public int lastIndexOf(Object o) { return list.lastIndexOf(o); }  public E set(int index, E element) {  return list.set(index, typeCheck(element));  }  public void add(int index, E element) {  list.add(index, typeCheck(element));  }  public boolean addAll(int index, Collection<? extends E> c) {  return list.addAll(index, checkedCopyOf(c));  }  public ListIterator<E> listIterator() { return listIterator(0); }  public ListIterator<E> listIterator(final int index) {  final ListIterator<E> i = list.listIterator(index);  return new ListIterator<E>() {  public boolean hasNext() { return i.hasNext(); }  public E next() { return i.next(); }  public boolean hasPrevious() { return i.hasPrevious(); }  public E previous() { return i.previous(); }  public int nextIndex() { return i.nextIndex(); }  public int previousIndex() { return i.previousIndex(); }  public void remove() { i.remove(); }  public void set(E e) {  i.set(typeCheck(e));  }  public void add(E e) {  i.add(typeCheck(e));  }  @Override  public void forEachRemaining(Consumer<? super E> action) {  i.forEachRemaining(action);  }  };  }  public List<E> subList(int fromIndex, int toIndex) {  return new CheckedList<>(list.subList(fromIndex, toIndex), type);  }  /\*\*  \* {@inheritDoc}  \*  \* @throws ClassCastException if the class of an element returned by the  \* operator prevents it from being added to this collection. The  \* exception may be thrown after some elements of the list have  \* already been replaced.  \*/  @Override  public void replaceAll(UnaryOperator<E> operator) {  Objects.requireNonNull(operator);  list.replaceAll(e -> typeCheck(operator.apply(e)));  }  @Override  public void sort(Comparator<? super E> c) {  list.sort(c);  }  }  /\*\*  \* @serial include  \*/  static class CheckedRandomAccessList<E> extends CheckedList<E>  implements RandomAccess  {  private static final long serialVersionUID = 1638200125423088369L;  CheckedRandomAccessList(List<E> list, Class<E> type) {  super(list, type);  }  public List<E> subList(int fromIndex, int toIndex) {  return new CheckedRandomAccessList<>(  list.subList(fromIndex, toIndex), type);  }  }  /\*\*  \* Returns a dynamically typesafe view of the specified map.  \* Any attempt to insert a mapping whose key or value have the wrong  \* type will result in an immediate {@link ClassCastException}.  \* Similarly, any attempt to modify the value currently associated with  \* a key will result in an immediate {@link ClassCastException},  \* whether the modification is attempted directly through the map  \* itself, or through a {@link Map.Entry} instance obtained from the  \* map's {@link Map#entrySet() entry set} view.  \*  \* <p>Assuming a map contains no incorrectly typed keys or values  \* prior to the time a dynamically typesafe view is generated, and  \* that all subsequent access to the map takes place through the view  \* (or one of its collection views), it is <i>guaranteed</i> that the  \* map cannot contain an incorrectly typed key or value.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned map will be serializable if the specified map is  \* serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned map permits insertion of null keys or values  \* whenever the backing map does.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the map for which a dynamically typesafe view is to be  \* returned  \* @param keyType the type of key that {@code m} is permitted to hold  \* @param valueType the type of value that {@code m} is permitted to hold  \* @return a dynamically typesafe view of the specified map  \* @since 1.5  \*/  public static <K, V> Map<K, V> checkedMap(Map<K, V> m,  Class<K> keyType,  Class<V> valueType) {  return new CheckedMap<>(m, keyType, valueType);  }  /\*\*  \* @serial include  \*/  private static class CheckedMap<K,V>  implements Map<K,V>, Serializable  {  private static final long serialVersionUID = 5742860141034234728L;  private final Map<K, V> m;  final Class<K> keyType;  final Class<V> valueType;  private void typeCheck(Object key, Object value) {  if (key != null && !keyType.isInstance(key))  throw new ClassCastException(badKeyMsg(key));  if (value != null && !valueType.isInstance(value))  throw new ClassCastException(badValueMsg(value));  }  private BiFunction<? super K, ? super V, ? extends V> typeCheck(  BiFunction<? super K, ? super V, ? extends V> func) {  Objects.requireNonNull(func);  return (k, v) -> {  V newValue = func.apply(k, v);  typeCheck(k, newValue);  return newValue;  };  }  private String badKeyMsg(Object key) {  return "Attempt to insert " + key.getClass() +  " key into map with key type " + keyType;  }  private String badValueMsg(Object value) {  return "Attempt to insert " + value.getClass() +  " value into map with value type " + valueType;  }  CheckedMap(Map<K, V> m, Class<K> keyType, Class<V> valueType) {  this.m = Objects.requireNonNull(m);  this.keyType = Objects.requireNonNull(keyType);  this.valueType = Objects.requireNonNull(valueType);  }  public int size() { return m.size(); }  public boolean isEmpty() { return m.isEmpty(); }  public boolean containsKey(Object key) { return m.containsKey(key); }  public boolean containsValue(Object v) { return m.containsValue(v); }  public V get(Object key) { return m.get(key); }  public V remove(Object key) { return m.remove(key); }  public void clear() { m.clear(); }  public Set<K> keySet() { return m.keySet(); }  public Collection<V> values() { return m.values(); }  public boolean equals(Object o) { return o == this || m.equals(o); }  public int hashCode() { return m.hashCode(); }  public String toString() { return m.toString(); }  public V put(K key, V value) {  typeCheck(key, value);  return m.put(key, value);  }  @SuppressWarnings("unchecked")  public void putAll(Map<? extends K, ? extends V> t) {  // Satisfy the following goals:  // - good diagnostics in case of type mismatch  // - all-or-nothing semantics  // - protection from malicious t  // - correct behavior if t is a concurrent map  Object[] entries = t.entrySet().toArray();  List<Map.Entry<K,V>> checked = new ArrayList<>(entries.length);  for (Object o : entries) {  Map.Entry<?,?> e = (Map.Entry<?,?>) o;  Object k = e.getKey();  Object v = e.getValue();  typeCheck(k, v);  checked.add(  new AbstractMap.SimpleImmutableEntry<>((K)k, (V)v));  }  for (Map.Entry<K,V> e : checked)  m.put(e.getKey(), e.getValue());  }  private transient Set<Map.Entry<K,V>> entrySet;  public Set<Map.Entry<K,V>> entrySet() {  if (entrySet==null)  entrySet = new CheckedEntrySet<>(m.entrySet(), valueType);  return entrySet;  }  // Override default methods in Map  @Override  public void forEach(BiConsumer<? super K, ? super V> action) {  m.forEach(action);  }  @Override  public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {  m.replaceAll(typeCheck(function));  }  @Override  public V putIfAbsent(K key, V value) {  typeCheck(key, value);  return m.putIfAbsent(key, value);  }  @Override  public boolean remove(Object key, Object value) {  return m.remove(key, value);  }  @Override  public boolean replace(K key, V oldValue, V newValue) {  typeCheck(key, newValue);  return m.replace(key, oldValue, newValue);  }  @Override  public V replace(K key, V value) {  typeCheck(key, value);  return m.replace(key, value);  }  @Override  public V computeIfAbsent(K key,  Function<? super K, ? extends V> mappingFunction) {  Objects.requireNonNull(mappingFunction);  return m.computeIfAbsent(key, k -> {  V value = mappingFunction.apply(k);  typeCheck(k, value);  return value;  });  }  @Override  public V computeIfPresent(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  return m.computeIfPresent(key, typeCheck(remappingFunction));  }  @Override  public V compute(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  return m.compute(key, typeCheck(remappingFunction));  }  @Override  public V merge(K key, V value,  BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  Objects.requireNonNull(remappingFunction);  return m.merge(key, value, (v1, v2) -> {  V newValue = remappingFunction.apply(v1, v2);  typeCheck(null, newValue);  return newValue;  });  }  /\*\*  \* We need this class in addition to CheckedSet as Map.Entry permits  \* modification of the backing Map via the setValue operation. This  \* class is subtle: there are many possible attacks that must be  \* thwarted.  \*  \* @serial exclude  \*/  static class CheckedEntrySet<K,V> implements Set<Map.Entry<K,V>> {  private final Set<Map.Entry<K,V>> s;  private final Class<V> valueType;  CheckedEntrySet(Set<Map.Entry<K, V>> s, Class<V> valueType) {  this.s = s;  this.valueType = valueType;  }  public int size() { return s.size(); }  public boolean isEmpty() { return s.isEmpty(); }  public String toString() { return s.toString(); }  public int hashCode() { return s.hashCode(); }  public void clear() { s.clear(); }  public boolean add(Map.Entry<K, V> e) {  throw new UnsupportedOperationException();  }  public boolean addAll(Collection<? extends Map.Entry<K, V>> coll) {  throw new UnsupportedOperationException();  }  public Iterator<Map.Entry<K,V>> iterator() {  final Iterator<Map.Entry<K, V>> i = s.iterator();  final Class<V> valueType = this.valueType;  return new Iterator<Map.Entry<K,V>>() {  public boolean hasNext() { return i.hasNext(); }  public void remove() { i.remove(); }  public Map.Entry<K,V> next() {  return checkedEntry(i.next(), valueType);  }  };  }  @SuppressWarnings("unchecked")  public Object[] toArray() {  Object[] source = s.toArray();  /\*  \* Ensure that we don't get an ArrayStoreException even if  \* s.toArray returns an array of something other than Object  \*/  Object[] dest = (CheckedEntry.class.isInstance(  source.getClass().getComponentType()) ? source :  new Object[source.length]);  for (int i = 0; i < source.length; i++)  dest[i] = checkedEntry((Map.Entry<K,V>)source[i],  valueType);  return dest;  }  @SuppressWarnings("unchecked")  public <T> T[] toArray(T[] a) {  // We don't pass a to s.toArray, to avoid window of  // vulnerability wherein an unscrupulous multithreaded client  // could get his hands on raw (unwrapped) Entries from s.  T[] arr = s.toArray(a.length==0 ? a : Arrays.copyOf(a, 0));  for (int i=0; i<arr.length; i++)  arr[i] = (T) checkedEntry((Map.Entry<K,V>)arr[i],  valueType);  if (arr.length > a.length)  return arr;  System.arraycopy(arr, 0, a, 0, arr.length);  if (a.length > arr.length)  a[arr.length] = null;  return a;  }  /\*\*  \* This method is overridden to protect the backing set against  \* an object with a nefarious equals function that senses  \* that the equality-candidate is Map.Entry and calls its  \* setValue method.  \*/  public boolean contains(Object o) {  if (!(o instanceof Map.Entry))  return false;  Map.Entry<?,?> e = (Map.Entry<?,?>) o;  return s.contains(  (e instanceof CheckedEntry) ? e : checkedEntry(e, valueType));  }  /\*\*  \* The bulk collection methods are overridden to protect  \* against an unscrupulous collection whose contains(Object o)  \* method senses when o is a Map.Entry, and calls o.setValue.  \*/  public boolean containsAll(Collection<?> c) {  for (Object o : c)  if (!contains(o)) // Invokes safe contains() above  return false;  return true;  }  public boolean remove(Object o) {  if (!(o instanceof Map.Entry))  return false;  return s.remove(new AbstractMap.SimpleImmutableEntry  <>((Map.Entry<?,?>)o));  }  public boolean removeAll(Collection<?> c) {  return batchRemove(c, false);  }  public boolean retainAll(Collection<?> c) {  return batchRemove(c, true);  }  private boolean batchRemove(Collection<?> c, boolean complement) {  Objects.requireNonNull(c);  boolean modified = false;  Iterator<Map.Entry<K,V>> it = iterator();  while (it.hasNext()) {  if (c.contains(it.next()) != complement) {  it.remove();  modified = true;  }  }  return modified;  }  public boolean equals(Object o) {  if (o == this)  return true;  if (!(o instanceof Set))  return false;  Set<?> that = (Set<?>) o;  return that.size() == s.size()  && containsAll(that); // Invokes safe containsAll() above  }  static <K,V,T> CheckedEntry<K,V,T> checkedEntry(Map.Entry<K,V> e,  Class<T> valueType) {  return new CheckedEntry<>(e, valueType);  }  /\*\*  \* This "wrapper class" serves two purposes: it prevents  \* the client from modifying the backing Map, by short-circuiting  \* the setValue method, and it protects the backing Map against  \* an ill-behaved Map.Entry that attempts to modify another  \* Map.Entry when asked to perform an equality check.  \*/  private static class CheckedEntry<K,V,T> implements Map.Entry<K,V> {  private final Map.Entry<K, V> e;  private final Class<T> valueType;  CheckedEntry(Map.Entry<K, V> e, Class<T> valueType) {  this.e = Objects.requireNonNull(e);  this.valueType = Objects.requireNonNull(valueType);  }  public K getKey() { return e.getKey(); }  public V getValue() { return e.getValue(); }  public int hashCode() { return e.hashCode(); }  public String toString() { return e.toString(); }  public V setValue(V value) {  if (value != null && !valueType.isInstance(value))  throw new ClassCastException(badValueMsg(value));  return e.setValue(value);  }  private String badValueMsg(Object value) {  return "Attempt to insert " + value.getClass() +  " value into map with value type " + valueType;  }  public boolean equals(Object o) {  if (o == this)  return true;  if (!(o instanceof Map.Entry))  return false;  return e.equals(new AbstractMap.SimpleImmutableEntry  <>((Map.Entry<?,?>)o));  }  }  }  }  /\*\*  \* Returns a dynamically typesafe view of the specified sorted map.  \* Any attempt to insert a mapping whose key or value have the wrong  \* type will result in an immediate {@link ClassCastException}.  \* Similarly, any attempt to modify the value currently associated with  \* a key will result in an immediate {@link ClassCastException},  \* whether the modification is attempted directly through the map  \* itself, or through a {@link Map.Entry} instance obtained from the  \* map's {@link Map#entrySet() entry set} view.  \*  \* <p>Assuming a map contains no incorrectly typed keys or values  \* prior to the time a dynamically typesafe view is generated, and  \* that all subsequent access to the map takes place through the view  \* (or one of its collection views), it is <i>guaranteed</i> that the  \* map cannot contain an incorrectly typed key or value.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned map will be serializable if the specified map is  \* serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned map permits insertion of null keys or values  \* whenever the backing map does.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param m the map for which a dynamically typesafe view is to be  \* returned  \* @param keyType the type of key that {@code m} is permitted to hold  \* @param valueType the type of value that {@code m} is permitted to hold  \* @return a dynamically typesafe view of the specified map  \* @since 1.5  \*/  public static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K, V> m,  Class<K> keyType,  Class<V> valueType) {  return new CheckedSortedMap<>(m, keyType, valueType);  }  /\*\*  \* @serial include  \*/  static class CheckedSortedMap<K,V> extends CheckedMap<K,V>  implements SortedMap<K,V>, Serializable  {  private static final long serialVersionUID = 1599671320688067438L;  private final SortedMap<K, V> sm;  CheckedSortedMap(SortedMap<K, V> m,  Class<K> keyType, Class<V> valueType) {  super(m, keyType, valueType);  sm = m;  }  public Comparator<? super K> comparator() { return sm.comparator(); }  public K firstKey() { return sm.firstKey(); }  public K lastKey() { return sm.lastKey(); }  public SortedMap<K,V> subMap(K fromKey, K toKey) {  return checkedSortedMap(sm.subMap(fromKey, toKey),  keyType, valueType);  }  public SortedMap<K,V> headMap(K toKey) {  return checkedSortedMap(sm.headMap(toKey), keyType, valueType);  }  public SortedMap<K,V> tailMap(K fromKey) {  return checkedSortedMap(sm.tailMap(fromKey), keyType, valueType);  }  }  /\*\*  \* Returns a dynamically typesafe view of the specified navigable map.  \* Any attempt to insert a mapping whose key or value have the wrong  \* type will result in an immediate {@link ClassCastException}.  \* Similarly, any attempt to modify the value currently associated with  \* a key will result in an immediate {@link ClassCastException},  \* whether the modification is attempted directly through the map  \* itself, or through a {@link Map.Entry} instance obtained from the  \* map's {@link Map#entrySet() entry set} view.  \*  \* <p>Assuming a map contains no incorrectly typed keys or values  \* prior to the time a dynamically typesafe view is generated, and  \* that all subsequent access to the map takes place through the view  \* (or one of its collection views), it is <em>guaranteed</em> that the  \* map cannot contain an incorrectly typed key or value.  \*  \* <p>A discussion of the use of dynamically typesafe views may be  \* found in the documentation for the {@link #checkedCollection  \* checkedCollection} method.  \*  \* <p>The returned map will be serializable if the specified map is  \* serializable.  \*  \* <p>Since {@code null} is considered to be a value of any reference  \* type, the returned map permits insertion of null keys or values  \* whenever the backing map does.  \*  \* @param <K> type of map keys  \* @param <V> type of map values  \* @param m the map for which a dynamically typesafe view is to be  \* returned  \* @param keyType the type of key that {@code m} is permitted to hold  \* @param valueType the type of value that {@code m} is permitted to hold  \* @return a dynamically typesafe view of the specified map  \* @since 1.8  \*/  public static <K,V> NavigableMap<K,V> checkedNavigableMap(NavigableMap<K, V> m,  Class<K> keyType,  Class<V> valueType) {  return new CheckedNavigableMap<>(m, keyType, valueType);  }  /\*\*  \* @serial include  \*/  static class CheckedNavigableMap<K,V> extends CheckedSortedMap<K,V>  implements NavigableMap<K,V>, Serializable  {  private static final long serialVersionUID = -4852462692372534096L;  private final NavigableMap<K, V> nm;  CheckedNavigableMap(NavigableMap<K, V> m,  Class<K> keyType, Class<V> valueType) {  super(m, keyType, valueType);  nm = m;  }  public Comparator<? super K> comparator() { return nm.comparator(); }  public K firstKey() { return nm.firstKey(); }  public K lastKey() { return nm.lastKey(); }  public Entry<K, V> lowerEntry(K key) {  Entry<K,V> lower = nm.lowerEntry(key);  return (null != lower)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(lower, valueType)  : null;  }  public K lowerKey(K key) { return nm.lowerKey(key); }  public Entry<K, V> floorEntry(K key) {  Entry<K,V> floor = nm.floorEntry(key);  return (null != floor)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(floor, valueType)  : null;  }  public K floorKey(K key) { return nm.floorKey(key); }  public Entry<K, V> ceilingEntry(K key) {  Entry<K,V> ceiling = nm.ceilingEntry(key);  return (null != ceiling)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(ceiling, valueType)  : null;  }  public K ceilingKey(K key) { return nm.ceilingKey(key); }  public Entry<K, V> higherEntry(K key) {  Entry<K,V> higher = nm.higherEntry(key);  return (null != higher)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(higher, valueType)  : null;  }  public K higherKey(K key) { return nm.higherKey(key); }  public Entry<K, V> firstEntry() {  Entry<K,V> first = nm.firstEntry();  return (null != first)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(first, valueType)  : null;  }  public Entry<K, V> lastEntry() {  Entry<K,V> last = nm.lastEntry();  return (null != last)  ? new CheckedMap.CheckedEntrySet.CheckedEntry<>(last, valueType)  : null;  }  public Entry<K, V> pollFirstEntry() {  Entry<K,V> entry = nm.pollFirstEntry();  return (null == entry)  ? null  : new CheckedMap.CheckedEntrySet.CheckedEntry<>(entry, valueType);  }  public Entry<K, V> pollLastEntry() {  Entry<K,V> entry = nm.pollLastEntry();  return (null == entry)  ? null  : new CheckedMap.CheckedEntrySet.CheckedEntry<>(entry, valueType);  }  public NavigableMap<K, V> descendingMap() {  return checkedNavigableMap(nm.descendingMap(), keyType, valueType);  }  public NavigableSet<K> keySet() {  return navigableKeySet();  }  public NavigableSet<K> navigableKeySet() {  return checkedNavigableSet(nm.navigableKeySet(), keyType);  }  public NavigableSet<K> descendingKeySet() {  return checkedNavigableSet(nm.descendingKeySet(), keyType);  }  @Override  public NavigableMap<K,V> subMap(K fromKey, K toKey) {  return checkedNavigableMap(nm.subMap(fromKey, true, toKey, false),  keyType, valueType);  }  @Override  public NavigableMap<K,V> headMap(K toKey) {  return checkedNavigableMap(nm.headMap(toKey, false), keyType, valueType);  }  @Override  public NavigableMap<K,V> tailMap(K fromKey) {  return checkedNavigableMap(nm.tailMap(fromKey, true), keyType, valueType);  }  public NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) {  return checkedNavigableMap(nm.subMap(fromKey, fromInclusive, toKey, toInclusive), keyType, valueType);  }  public NavigableMap<K, V> headMap(K toKey, boolean inclusive) {  return checkedNavigableMap(nm.headMap(toKey, inclusive), keyType, valueType);  }  public NavigableMap<K, V> tailMap(K fromKey, boolean inclusive) {  return checkedNavigableMap(nm.tailMap(fromKey, inclusive), keyType, valueType);  }  }  // Empty collections  /\*\*  \* Returns an iterator that has no elements. More precisely,  \*  \* <ul>  \* <li>{@link Iterator#hasNext hasNext} always returns {@code  \* false}.</li>  \* <li>{@link Iterator#next next} always throws {@link  \* NoSuchElementException}.</li>  \* <li>{@link Iterator#remove remove} always throws {@link  \* IllegalStateException}.</li>  \* </ul>  \*  \* <p>Implementations of this method are permitted, but not  \* required, to return the same object from multiple invocations.  \*  \* @param <T> type of elements, if there were any, in the iterator  \* @return an empty iterator  \* @since 1.7  \*/  @SuppressWarnings("unchecked")  public static <T> Iterator<T> emptyIterator() {  return (Iterator<T>) EmptyIterator.EMPTY\_ITERATOR;  }  private static class EmptyIterator<E> implements Iterator<E> {  static final EmptyIterator<Object> EMPTY\_ITERATOR  = new EmptyIterator<>();  public boolean hasNext() { return false; }  public E next() { throw new NoSuchElementException(); }  public void remove() { throw new IllegalStateException(); }  @Override  public void forEachRemaining(Consumer<? super E> action) {  Objects.requireNonNull(action);  }  }  /\*\*  \* Returns a list iterator that has no elements. More precisely,  \*  \* <ul>  \* <li>{@link Iterator#hasNext hasNext} and {@link  \* ListIterator#hasPrevious hasPrevious} always return {@code  \* false}.</li>  \* <li>{@link Iterator#next next} and {@link ListIterator#previous  \* previous} always throw {@link NoSuchElementException}.</li>  \* <li>{@link Iterator#remove remove} and {@link ListIterator#set  \* set} always throw {@link IllegalStateException}.</li>  \* <li>{@link ListIterator#add add} always throws {@link  \* UnsupportedOperationException}.</li>  \* <li>{@link ListIterator#nextIndex nextIndex} always returns  \* {@code 0}.</li>  \* <li>{@link ListIterator#previousIndex previousIndex} always  \* returns {@code -1}.</li>  \* </ul>  \*  \* <p>Implementations of this method are permitted, but not  \* required, to return the same object from multiple invocations.  \*  \* @param <T> type of elements, if there were any, in the iterator  \* @return an empty list iterator  \* @since 1.7  \*/  @SuppressWarnings("unchecked")  public static <T> ListIterator<T> emptyListIterator() {  return (ListIterator<T>) EmptyListIterator.EMPTY\_ITERATOR;  }  private static class EmptyListIterator<E>  extends EmptyIterator<E>  implements ListIterator<E>  {  static final EmptyListIterator<Object> EMPTY\_ITERATOR  = new EmptyListIterator<>();  public boolean hasPrevious() { return false; }  public E previous() { throw new NoSuchElementException(); }  public int nextIndex() { return 0; }  public int previousIndex() { return -1; }  public void set(E e) { throw new IllegalStateException(); }  public void add(E e) { throw new UnsupportedOperationException(); }  }  /\*\*  \* Returns an enumeration that has no elements. More precisely,  \*  \* <ul>  \* <li>{@link Enumeration#hasMoreElements hasMoreElements} always  \* returns {@code false}.</li>  \* <li> {@link Enumeration#nextElement nextElement} always throws  \* {@link NoSuchElementException}.</li>  \* </ul>  \*  \* <p>Implementations of this method are permitted, but not  \* required, to return the same object from multiple invocations.  \*  \* @param <T> the class of the objects in the enumeration  \* @return an empty enumeration  \* @since 1.7  \*/  @SuppressWarnings("unchecked")  public static <T> Enumeration<T> emptyEnumeration() {  return (Enumeration<T>) EmptyEnumeration.EMPTY\_ENUMERATION;  }  private static class EmptyEnumeration<E> implements Enumeration<E> {  static final EmptyEnumeration<Object> EMPTY\_ENUMERATION  = new EmptyEnumeration<>();  public boolean hasMoreElements() { return false; }  public E nextElement() { throw new NoSuchElementException(); }  }  /\*\*  \* The empty set (immutable). This set is serializable.  \*  \* @see #emptySet()  \*/  @SuppressWarnings("rawtypes")  public static final Set EMPTY\_SET = new EmptySet<>();  /\*\*  \* Returns an empty set (immutable). This set is serializable.  \* Unlike the like-named field, this method is parameterized.  \*  \* <p>This example illustrates the type-safe way to obtain an empty set:  \* <pre>  \* Set&lt;String&gt; s = Collections.emptySet();  \* </pre>  \* @implNote Implementations of this method need not create a separate  \* {@code Set} object for each call. Using this method is likely to have  \* comparable cost to using the like-named field. (Unlike this method, the  \* field does not provide type safety.)  \*  \* @param <T> the class of the objects in the set  \* @return the empty set  \*  \* @see #EMPTY\_SET  \* @since 1.5  \*/  @SuppressWarnings("unchecked")  public static final <T> Set<T> emptySet() {  return (Set<T>) EMPTY\_SET;  }  /\*\*  \* @serial include  \*/  private static class EmptySet<E>  extends AbstractSet<E>  implements Serializable  {  private static final long serialVersionUID = 1582296315990362920L;  public Iterator<E> iterator() { return emptyIterator(); }  public int size() {return 0;}  public boolean isEmpty() {return true;}  public boolean contains(Object obj) {return false;}  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }  public Object[] toArray() { return new Object[0]; }  public <T> T[] toArray(T[] a) {  if (a.length > 0)  a[0] = null;  return a;  }  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {  Objects.requireNonNull(action);  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  Objects.requireNonNull(filter);  return false;  }  @Override  public Spliterator<E> spliterator() { return Spliterators.emptySpliterator(); }  // Preserves singleton property  private Object readResolve() {  return EMPTY\_SET;  }  }  /\*\*  \* Returns an empty sorted set (immutable). This set is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty  \* sorted set:  \* <pre> {@code  \* SortedSet<String> s = Collections.emptySortedSet();  \* }</pre>  \*  \* @implNote Implementations of this method need not create a separate  \* {@code SortedSet} object for each call.  \*  \* @param <E> type of elements, if there were any, in the set  \* @return the empty sorted set  \* @since 1.8  \*/  @SuppressWarnings("unchecked")  public static <E> SortedSet<E> emptySortedSet() {  return (SortedSet<E>) UnmodifiableNavigableSet.EMPTY\_NAVIGABLE\_SET;  }  /\*\*  \* Returns an empty navigable set (immutable). This set is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty  \* navigable set:  \* <pre> {@code  \* NavigableSet<String> s = Collections.emptyNavigableSet();  \* }</pre>  \*  \* @implNote Implementations of this method need not  \* create a separate {@code NavigableSet} object for each call.  \*  \* @param <E> type of elements, if there were any, in the set  \* @return the empty navigable set  \* @since 1.8  \*/  @SuppressWarnings("unchecked")  public static <E> NavigableSet<E> emptyNavigableSet() {  return (NavigableSet<E>) UnmodifiableNavigableSet.EMPTY\_NAVIGABLE\_SET;  }  /\*\*  \* The empty list (immutable). This list is serializable.  \*  \* @see #emptyList()  \*/  @SuppressWarnings("rawtypes")  public static final List EMPTY\_LIST = new EmptyList<>();  /\*\*  \* Returns an empty list (immutable). This list is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty list:  \* <pre>  \* List&lt;String&gt; s = Collections.emptyList();  \* </pre>  \*  \* @implNote  \* Implementations of this method need not create a separate <tt>List</tt>  \* object for each call. Using this method is likely to have comparable  \* cost to using the like-named field. (Unlike this method, the field does  \* not provide type safety.)  \*  \* @param <T> type of elements, if there were any, in the list  \* @return an empty immutable list  \*  \* @see #EMPTY\_LIST  \* @since 1.5  \*/  @SuppressWarnings("unchecked")  public static final <T> List<T> emptyList() {  return (List<T>) EMPTY\_LIST;  }  /\*\*  \* @serial include  \*/  private static class EmptyList<E>  extends AbstractList<E>  implements RandomAccess, Serializable {  private static final long serialVersionUID = 8842843931221139166L;  public Iterator<E> iterator() {  return emptyIterator();  }  public ListIterator<E> listIterator() {  return emptyListIterator();  }  public int size() {return 0;}  public boolean isEmpty() {return true;}  public boolean contains(Object obj) {return false;}  public boolean containsAll(Collection<?> c) { return c.isEmpty(); }  public Object[] toArray() { return new Object[0]; }  public <T> T[] toArray(T[] a) {  if (a.length > 0)  a[0] = null;  return a;  }  public E get(int index) {  throw new IndexOutOfBoundsException("Index: "+index);  }  public boolean equals(Object o) {  return (o instanceof List) && ((List<?>)o).isEmpty();  }  public int hashCode() { return 1; }  @Override  public boolean removeIf(Predicate<? super E> filter) {  Objects.requireNonNull(filter);  return false;  }  @Override  public void replaceAll(UnaryOperator<E> operator) {  Objects.requireNonNull(operator);  }  @Override  public void sort(Comparator<? super E> c) {  }  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {  Objects.requireNonNull(action);  }  @Override  public Spliterator<E> spliterator() { return Spliterators.emptySpliterator(); }  // Preserves singleton property  private Object readResolve() {  return EMPTY\_LIST;  }  }  /\*\*  \* The empty map (immutable). This map is serializable.  \*  \* @see #emptyMap()  \* @since 1.3  \*/  @SuppressWarnings("rawtypes")  public static final Map EMPTY\_MAP = new EmptyMap<>();  /\*\*  \* Returns an empty map (immutable). This map is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty map:  \* <pre>  \* Map&lt;String, Date&gt; s = Collections.emptyMap();  \* </pre>  \* @implNote Implementations of this method need not create a separate  \* {@code Map} object for each call. Using this method is likely to have  \* comparable cost to using the like-named field. (Unlike this method, the  \* field does not provide type safety.)  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @return an empty map  \* @see #EMPTY\_MAP  \* @since 1.5  \*/  @SuppressWarnings("unchecked")  public static final <K,V> Map<K,V> emptyMap() {  return (Map<K,V>) EMPTY\_MAP;  }  /\*\*  \* Returns an empty sorted map (immutable). This map is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty map:  \* <pre> {@code  \* SortedMap<String, Date> s = Collections.emptySortedMap();  \* }</pre>  \*  \* @implNote Implementations of this method need not create a separate  \* {@code SortedMap} object for each call.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @return an empty sorted map  \* @since 1.8  \*/  @SuppressWarnings("unchecked")  public static final <K,V> SortedMap<K,V> emptySortedMap() {  return (SortedMap<K,V>) UnmodifiableNavigableMap.EMPTY\_NAVIGABLE\_MAP;  }  /\*\*  \* Returns an empty navigable map (immutable). This map is serializable.  \*  \* <p>This example illustrates the type-safe way to obtain an empty map:  \* <pre> {@code  \* NavigableMap<String, Date> s = Collections.emptyNavigableMap();  \* }</pre>  \*  \* @implNote Implementations of this method need not create a separate  \* {@code NavigableMap} object for each call.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @return an empty navigable map  \* @since 1.8  \*/  @SuppressWarnings("unchecked")  public static final <K,V> NavigableMap<K,V> emptyNavigableMap() {  return (NavigableMap<K,V>) UnmodifiableNavigableMap.EMPTY\_NAVIGABLE\_MAP;  }  /\*\*  \* @serial include  \*/  private static class EmptyMap<K,V>  extends AbstractMap<K,V>  implements Serializable  {  private static final long serialVersionUID = 6428348081105594320L;  public int size() {return 0;}  public boolean isEmpty() {return true;}  public boolean containsKey(Object key) {return false;}  public boolean containsValue(Object value) {return false;}  public V get(Object key) {return null;}  public Set<K> keySet() {return emptySet();}  public Collection<V> values() {return emptySet();}  public Set<Map.Entry<K,V>> entrySet() {return emptySet();}  public boolean equals(Object o) {  return (o instanceof Map) && ((Map<?,?>)o).isEmpty();  }  public int hashCode() {return 0;}  // Override default methods in Map  @Override  @SuppressWarnings("unchecked")  public V getOrDefault(Object k, V defaultValue) {  return defaultValue;  }  @Override  public void forEach(BiConsumer<? super K, ? super V> action) {  Objects.requireNonNull(action);  }  @Override  public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {  Objects.requireNonNull(function);  }  @Override  public V putIfAbsent(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public boolean remove(Object key, Object value) {  throw new UnsupportedOperationException();  }  @Override  public boolean replace(K key, V oldValue, V newValue) {  throw new UnsupportedOperationException();  }  @Override  public V replace(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfAbsent(K key,  Function<? super K, ? extends V> mappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfPresent(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V compute(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V merge(K key, V value,  BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  // Preserves singleton property  private Object readResolve() {  return EMPTY\_MAP;  }  }  // Singleton collections  /\*\*  \* Returns an immutable set containing only the specified object.  \* The returned set is serializable.  \*  \* @param <T> the class of the objects in the set  \* @param o the sole object to be stored in the returned set.  \* @return an immutable set containing only the specified object.  \*/  public static <T> Set<T> singleton(T o) {  return new SingletonSet<>(o);  }  static <E> Iterator<E> singletonIterator(final E e) {  return new Iterator<E>() {  private boolean hasNext = true;  public boolean hasNext() {  return hasNext;  }  public E next() {  if (hasNext) {  hasNext = false;  return e;  }  throw new NoSuchElementException();  }  public void remove() {  throw new UnsupportedOperationException();  }  @Override  public void forEachRemaining(Consumer<? super E> action) {  Objects.requireNonNull(action);  if (hasNext) {  action.accept(e);  hasNext = false;  }  }  };  }  /\*\*  \* Creates a {@code Spliterator} with only the specified element  \*  \* @param <T> Type of elements  \* @return A singleton {@code Spliterator}  \*/  static <T> Spliterator<T> singletonSpliterator(final T element) {  return new Spliterator<T>() {  long est = 1;  @Override  public Spliterator<T> trySplit() {  return null;  }  @Override  public boolean tryAdvance(Consumer<? super T> consumer) {  Objects.requireNonNull(consumer);  if (est > 0) {  est--;  consumer.accept(element);  return true;  }  return false;  }  @Override  public void forEachRemaining(Consumer<? super T> consumer) {  tryAdvance(consumer);  }  @Override  public long estimateSize() {  return est;  }  @Override  public int characteristics() {  int value = (element != null) ? Spliterator.NONNULL : 0;  return value | Spliterator.SIZED | Spliterator.SUBSIZED | Spliterator.IMMUTABLE |  Spliterator.DISTINCT | Spliterator.ORDERED;  }  };  }  /\*\*  \* @serial include  \*/  private static class SingletonSet<E>  extends AbstractSet<E>  implements Serializable  {  private static final long serialVersionUID = 3193687207550431679L;  private final E element;  SingletonSet(E e) {element = e;}  public Iterator<E> iterator() {  return singletonIterator(element);  }  public int size() {return 1;}  public boolean contains(Object o) {return eq(o, element);}  // Override default methods for Collection  @Override  public void forEach(Consumer<? super E> action) {  action.accept(element);  }  @Override  public Spliterator<E> spliterator() {  return singletonSpliterator(element);  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  throw new UnsupportedOperationException();  }  }  /\*\*  \* Returns an immutable list containing only the specified object.  \* The returned list is serializable.  \*  \* @param <T> the class of the objects in the list  \* @param o the sole object to be stored in the returned list.  \* @return an immutable list containing only the specified object.  \* @since 1.3  \*/  public static <T> List<T> singletonList(T o) {  return new SingletonList<>(o);  }  /\*\*  \* @serial include  \*/  private static class SingletonList<E>  extends AbstractList<E>  implements RandomAccess, Serializable {  private static final long serialVersionUID = 3093736618740652951L;  private final E element;  SingletonList(E obj) {element = obj;}  public Iterator<E> iterator() {  return singletonIterator(element);  }  public int size() {return 1;}  public boolean contains(Object obj) {return eq(obj, element);}  public E get(int index) {  if (index != 0)  throw new IndexOutOfBoundsException("Index: "+index+", Size: 1");  return element;  }  // Override default methods for Collection  @Override  public void forEach(Consumer<? super E> action) {  action.accept(element);  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  throw new UnsupportedOperationException();  }  @Override  public void replaceAll(UnaryOperator<E> operator) {  throw new UnsupportedOperationException();  }  @Override  public void sort(Comparator<? super E> c) {  }  @Override  public Spliterator<E> spliterator() {  return singletonSpliterator(element);  }  }  /\*\*  \* Returns an immutable map, mapping only the specified key to the  \* specified value. The returned map is serializable.  \*  \* @param <K> the class of the map keys  \* @param <V> the class of the map values  \* @param key the sole key to be stored in the returned map.  \* @param value the value to which the returned map maps <tt>key</tt>.  \* @return an immutable map containing only the specified key-value  \* mapping.  \* @since 1.3  \*/  public static <K,V> Map<K,V> singletonMap(K key, V value) {  return new SingletonMap<>(key, value);  }  /\*\*  \* @serial include  \*/  private static class SingletonMap<K,V>  extends AbstractMap<K,V>  implements Serializable {  private static final long serialVersionUID = -6979724477215052911L;  private final K k;  private final V v;  SingletonMap(K key, V value) {  k = key;  v = value;  }  public int size() {return 1;}  public boolean isEmpty() {return false;}  public boolean containsKey(Object key) {return eq(key, k);}  public boolean containsValue(Object value) {return eq(value, v);}  public V get(Object key) {return (eq(key, k) ? v : null);}  private transient Set<K> keySet;  private transient Set<Map.Entry<K,V>> entrySet;  private transient Collection<V> values;  public Set<K> keySet() {  if (keySet==null)  keySet = singleton(k);  return keySet;  }  public Set<Map.Entry<K,V>> entrySet() {  if (entrySet==null)  entrySet = Collections.<Map.Entry<K,V>>singleton(  new SimpleImmutableEntry<>(k, v));  return entrySet;  }  public Collection<V> values() {  if (values==null)  values = singleton(v);  return values;  }  // Override default methods in Map  @Override  public V getOrDefault(Object key, V defaultValue) {  return eq(key, k) ? v : defaultValue;  }  @Override  public void forEach(BiConsumer<? super K, ? super V> action) {  action.accept(k, v);  }  @Override  public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {  throw new UnsupportedOperationException();  }  @Override  public V putIfAbsent(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public boolean remove(Object key, Object value) {  throw new UnsupportedOperationException();  }  @Override  public boolean replace(K key, V oldValue, V newValue) {  throw new UnsupportedOperationException();  }  @Override  public V replace(K key, V value) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfAbsent(K key,  Function<? super K, ? extends V> mappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V computeIfPresent(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V compute(K key,  BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  @Override  public V merge(K key, V value,  BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  throw new UnsupportedOperationException();  }  }  // Miscellaneous  /\*\*  \* Returns an immutable list consisting of <tt>n</tt> copies of the  \* specified object. The newly allocated data object is tiny (it contains  \* a single reference to the data object). This method is useful in  \* combination with the <tt>List.addAll</tt> method to grow lists.  \* The returned list is serializable.  \*  \* @param <T> the class of the object to copy and of the objects  \* in the returned list.  \* @param n the number of elements in the returned list.  \* @param o the element to appear repeatedly in the returned list.  \* @return an immutable list consisting of <tt>n</tt> copies of the  \* specified object.  \* @throws IllegalArgumentException if {@code n < 0}  \* @see List#addAll(Collection)  \* @see List#addAll(int, Collection)  \*/  public static <T> List<T> nCopies(int n, T o) {  if (n < 0)  throw new IllegalArgumentException("List length = " + n);  return new CopiesList<>(n, o);  }  /\*\*  \* @serial include  \*/  private static class CopiesList<E>  extends AbstractList<E>  implements RandomAccess, Serializable  {  private static final long serialVersionUID = 2739099268398711800L;  final int n;  final E element;  CopiesList(int n, E e) {  assert n >= 0;  this.n = n;  element = e;  }  public int size() {  return n;  }  public boolean contains(Object obj) {  return n != 0 && eq(obj, element);  }  public int indexOf(Object o) {  return contains(o) ? 0 : -1;  }  public int lastIndexOf(Object o) {  return contains(o) ? n - 1 : -1;  }  public E get(int index) {  if (index < 0 || index >= n)  throw new IndexOutOfBoundsException("Index: "+index+  ", Size: "+n);  return element;  }  public Object[] toArray() {  final Object[] a = new Object[n];  if (element != null)  Arrays.fill(a, 0, n, element);  return a;  }  @SuppressWarnings("unchecked")  public <T> T[] toArray(T[] a) {  final int n = this.n;  if (a.length < n) {  a = (T[])java.lang.reflect.Array  .newInstance(a.getClass().getComponentType(), n);  if (element != null)  Arrays.fill(a, 0, n, element);  } else {  Arrays.fill(a, 0, n, element);  if (a.length > n)  a[n] = null;  }  return a;  }  public List<E> subList(int fromIndex, int toIndex) {  if (fromIndex < 0)  throw new IndexOutOfBoundsException("fromIndex = " + fromIndex);  if (toIndex > n)  throw new IndexOutOfBoundsException("toIndex = " + toIndex);  if (fromIndex > toIndex)  throw new IllegalArgumentException("fromIndex(" + fromIndex +  ") > toIndex(" + toIndex + ")");  return new CopiesList<>(toIndex - fromIndex, element);  }  @Override  public int hashCode() {  if (n == 0) return 1;  // hashCode of n repeating elements is 31^n + elementHash \* Sum(31^k, k = 0..n-1)  // this implementation completes in O(log(n)) steps taking advantage of  // 31^(2\*n) = (31^n)^2 and Sum(31^k, k = 0..(2\*n-1)) = Sum(31^k, k = 0..n-1) \* (31^n + 1)  int pow = 31;  int sum = 1;  for (int i = Integer.numberOfLeadingZeros(n) + 1; i < Integer.SIZE; i++) {  sum \*= pow + 1;  pow \*= pow;  if ((n << i) < 0) {  pow \*= 31;  sum = sum \* 31 + 1;  }  }  return pow + sum \* (element == null ? 0 : element.hashCode());  }  @Override  public boolean equals(Object o) {  if (o == this)  return true;  if (o instanceof CopiesList) {  CopiesList<?> other = (CopiesList<?>) o;  return n == other.n && (n == 0 || eq(element, other.element));  }  if (!(o instanceof List))  return false;  int remaining = n;  E e = element;  Iterator<?> itr = ((List<?>) o).iterator();  if (e == null) {  while (itr.hasNext() && remaining-- > 0) {  if (itr.next() != null)  return false;  }  } else {  while (itr.hasNext() && remaining-- > 0) {  if (!e.equals(itr.next()))  return false;  }  }  return remaining == 0 && !itr.hasNext();  }  // Override default methods in Collection  @Override  public Stream<E> stream() {  return IntStream.range(0, n).mapToObj(i -> element);  }  @Override  public Stream<E> parallelStream() {  return IntStream.range(0, n).parallel().mapToObj(i -> element);  }  @Override  public Spliterator<E> spliterator() {  return stream().spliterator();  }  private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {  ois.defaultReadObject();  SharedSecrets.getJavaOISAccess().checkArray(ois, Object[].class, n);  }  }  /\*\*  \* Returns a comparator that imposes the reverse of the <em>natural  \* ordering</em> on a collection of objects that implement the  \* {@code Comparable} interface. (The natural ordering is the ordering  \* imposed by the objects' own {@code compareTo} method.) This enables a  \* simple idiom for sorting (or maintaining) collections (or arrays) of  \* objects that implement the {@code Comparable} interface in  \* reverse-natural-order. For example, suppose {@code a} is an array of  \* strings. Then: <pre>  \* Arrays.sort(a, Collections.reverseOrder());  \* </pre> sorts the array in reverse-lexicographic (alphabetical) order.<p>  \*  \* The returned comparator is serializable.  \*  \* @param <T> the class of the objects compared by the comparator  \* @return A comparator that imposes the reverse of the <i>natural  \* ordering</i> on a collection of objects that implement  \* the <tt>Comparable</tt> interface.  \* @see Comparable  \*/  @SuppressWarnings("unchecked")  public static <T> Comparator<T> reverseOrder() {  return (Comparator<T>) ReverseComparator.REVERSE\_ORDER;  }  /\*\*  \* @serial include  \*/  private static class ReverseComparator  implements Comparator<Comparable<Object>>, Serializable {  private static final long serialVersionUID = 7207038068494060240L;  static final ReverseComparator REVERSE\_ORDER  = new ReverseComparator();  public int compare(Comparable<Object> c1, Comparable<Object> c2) {  return c2.compareTo(c1);  }  private Object readResolve() { return Collections.reverseOrder(); }  @Override  public Comparator<Comparable<Object>> reversed() {  return Comparator.naturalOrder();  }  }  /\*\*  \* Returns a comparator that imposes the reverse ordering of the specified  \* comparator. If the specified comparator is {@code null}, this method is  \* equivalent to {@link #reverseOrder()} (in other words, it returns a  \* comparator that imposes the reverse of the <em>natural ordering</em> on  \* a collection of objects that implement the Comparable interface).  \*  \* <p>The returned comparator is serializable (assuming the specified  \* comparator is also serializable or {@code null}).  \*  \* @param <T> the class of the objects compared by the comparator  \* @param cmp a comparator who's ordering is to be reversed by the returned  \* comparator or {@code null}  \* @return A comparator that imposes the reverse ordering of the  \* specified comparator.  \* @since 1.5  \*/  public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {  if (cmp == null)  return reverseOrder();  if (cmp instanceof ReverseComparator2)  return ((ReverseComparator2<T>)cmp).cmp;  return new ReverseComparator2<>(cmp);  }  /\*\*  \* @serial include  \*/  private static class ReverseComparator2<T> implements Comparator<T>,  Serializable  {  private static final long serialVersionUID = 4374092139857L;  /\*\*  \* The comparator specified in the static factory. This will never  \* be null, as the static factory returns a ReverseComparator  \* instance if its argument is null.  \*  \* @serial  \*/  final Comparator<T> cmp;  ReverseComparator2(Comparator<T> cmp) {  assert cmp != null;  this.cmp = cmp;  }  public int compare(T t1, T t2) {  return cmp.compare(t2, t1);  }  public boolean equals(Object o) {  return (o == this) ||  (o instanceof ReverseComparator2 &&  cmp.equals(((ReverseComparator2)o).cmp));  }  public int hashCode() {  return cmp.hashCode() ^ Integer.MIN\_VALUE;  }  @Override  public Comparator<T> reversed() {  return cmp;  }  }  /\*\*  \* Returns an enumeration over the specified collection. This provides  \* interoperability with legacy APIs that require an enumeration  \* as input.  \*  \* @param <T> the class of the objects in the collection  \* @param c the collection for which an enumeration is to be returned.  \* @return an enumeration over the specified collection.  \* @see Enumeration  \*/  public static <T> Enumeration<T> enumeration(final Collection<T> c) {  return new Enumeration<T>() {  private final Iterator<T> i = c.iterator();  public boolean hasMoreElements() {  return i.hasNext();  }  public T nextElement() {  return i.next();  }  };  }  /\*\*  \* Returns an array list containing the elements returned by the  \* specified enumeration in the order they are returned by the  \* enumeration. This method provides interoperability between  \* legacy APIs that return enumerations and new APIs that require  \* collections.  \*  \* @param <T> the class of the objects returned by the enumeration  \* @param e enumeration providing elements for the returned  \* array list  \* @return an array list containing the elements returned  \* by the specified enumeration.  \* @since 1.4  \* @see Enumeration  \* @see ArrayList  \*/  public static <T> ArrayList<T> list(Enumeration<T> e) {  ArrayList<T> l = new ArrayList<>();  while (e.hasMoreElements())  l.add(e.nextElement());  return l;  }  /\*\*  \* Returns true if the specified arguments are equal, or both null.  \*  \* NB: Do not replace with Object.equals until JDK-8015417 is resolved.  \*/  static boolean eq(Object o1, Object o2) {  return o1==null ? o2==null : o1.equals(o2);  }  /\*\*  \* Returns the number of elements in the specified collection equal to the  \* specified object. More formally, returns the number of elements  \* <tt>e</tt> in the collection such that  \* <tt>(o == null ? e == null : o.equals(e))</tt>.  \*  \* @param c the collection in which to determine the frequency  \* of <tt>o</tt>  \* @param o the object whose frequency is to be determined  \* @return the number of elements in {@code c} equal to {@code o}  \* @throws NullPointerException if <tt>c</tt> is null  \* @since 1.5  \*/  public static int frequency(Collection<?> c, Object o) {  int result = 0;  if (o == null) {  for (Object e : c)  if (e == null)  result++;  } else {  for (Object e : c)  if (o.equals(e))  result++;  }  return result;  }  /\*\*  \* Returns {@code true} if the two specified collections have no  \* elements in common.  \*  \* <p>Care must be exercised if this method is used on collections that  \* do not comply with the general contract for {@code Collection}.  \* Implementations may elect to iterate over either collection and test  \* for containment in the other collection (or to perform any equivalent  \* computation). If either collection uses a nonstandard equality test  \* (as does a {@link SortedSet} whose ordering is not <em>compatible with  \* equals</em>, or the key set of an {@link IdentityHashMap}), both  \* collections must use the same nonstandard equality test, or the  \* result of this method is undefined.  \*  \* <p>Care must also be exercised when using collections that have  \* restrictions on the elements that they may contain. Collection  \* implementations are allowed to throw exceptions for any operation  \* involving elements they deem ineligible. For absolute safety the  \* specified collections should contain only elements which are  \* eligible elements for both collections.  \*  \* <p>Note that it is permissible to pass the same collection in both  \* parameters, in which case the method will return {@code true} if and  \* only if the collection is empty.  \*  \* @param c1 a collection  \* @param c2 a collection  \* @return {@code true} if the two specified collections have no  \* elements in common.  \* @throws NullPointerException if either collection is {@code null}.  \* @throws NullPointerException if one collection contains a {@code null}  \* element and {@code null} is not an eligible element for the other collection.  \* (<a href="Collection.html#optional-restrictions">optional</a>)  \* @throws ClassCastException if one collection contains an element that is  \* of a type which is ineligible for the other collection.  \* (<a href="Collection.html#optional-restrictions">optional</a>)  \* @since 1.5  \*/  public static boolean disjoint(Collection<?> c1, Collection<?> c2) {  // The collection to be used for contains(). Preference is given to  // the collection who's contains() has lower O() complexity.  Collection<?> contains = c2;  // The collection to be iterated. If the collections' contains() impl  // are of different O() complexity, the collection with slower  // contains() will be used for iteration. For collections who's  // contains() are of the same complexity then best performance is  // achieved by iterating the smaller collection.  Collection<?> iterate = c1;  // Performance optimization cases. The heuristics:  // 1. Generally iterate over c1.  // 2. If c1 is a Set then iterate over c2.  // 3. If either collection is empty then result is always true.  // 4. Iterate over the smaller Collection.  if (c1 instanceof Set) {  // Use c1 for contains as a Set's contains() is expected to perform  // better than O(N/2)  iterate = c2;  contains = c1;  } else if (!(c2 instanceof Set)) {  // Both are mere Collections. Iterate over smaller collection.  // Example: If c1 contains 3 elements and c2 contains 50 elements and  // assuming contains() requires ceiling(N/2) comparisons then  // checking for all c1 elements in c2 would require 75 comparisons  // (3 \* ceiling(50/2)) vs. checking all c2 elements in c1 requiring  // 100 comparisons (50 \* ceiling(3/2)).  int c1size = c1.size();  int c2size = c2.size();  if (c1size == 0 || c2size == 0) {  // At least one collection is empty. Nothing will match.  return true;  }  if (c1size > c2size) {  iterate = c2;  contains = c1;  }  }  for (Object e : iterate) {  if (contains.contains(e)) {  // Found a common element. Collections are not disjoint.  return false;  }  }  // No common elements were found.  return true;  }  /\*\*  \* Adds all of the specified elements to the specified collection.  \* Elements to be added may be specified individually or as an array.  \* The behavior of this convenience method is identical to that of  \* <tt>c.addAll(Arrays.asList(elements))</tt>, but this method is likely  \* to run significantly faster under most implementations.  \*  \* <p>When elements are specified individually, this method provides a  \* convenient way to add a few elements to an existing collection:  \* <pre>  \* Collections.addAll(flavors, "Peaches 'n Plutonium", "Rocky Racoon");  \* </pre>  \*  \* @param <T> the class of the elements to add and of the collection  \* @param c the collection into which <tt>elements</tt> are to be inserted  \* @param elements the elements to insert into <tt>c</tt>  \* @return <tt>true</tt> if the collection changed as a result of the call  \* @throws UnsupportedOperationException if <tt>c</tt> does not support  \* the <tt>add</tt> operation  \* @throws NullPointerException if <tt>elements</tt> contains one or more  \* null values and <tt>c</tt> does not permit null elements, or  \* if <tt>c</tt> or <tt>elements</tt> are <tt>null</tt>  \* @throws IllegalArgumentException if some property of a value in  \* <tt>elements</tt> prevents it from being added to <tt>c</tt>  \* @see Collection#addAll(Collection)  \* @since 1.5  \*/  @SafeVarargs  public static <T> boolean addAll(Collection<? super T> c, T... elements) {  boolean result = false;  for (T element : elements)  result |= c.add(element);  return result;  }  /\*\*  \* Returns a set backed by the specified map. The resulting set displays  \* the same ordering, concurrency, and performance characteristics as the  \* backing map. In essence, this factory method provides a {@link Set}  \* implementation corresponding to any {@link Map} implementation. There  \* is no need to use this method on a {@link Map} implementation that  \* already has a corresponding {@link Set} implementation (such as {@link  \* HashMap} or {@link TreeMap}).  \*  \* <p>Each method invocation on the set returned by this method results in  \* exactly one method invocation on the backing map or its <tt>keySet</tt>  \* view, with one exception. The <tt>addAll</tt> method is implemented  \* as a sequence of <tt>put</tt> invocations on the backing map.  \*  \* <p>The specified map must be empty at the time this method is invoked,  \* and should not be accessed directly after this method returns. These  \* conditions are ensured if the map is created empty, passed directly  \* to this method, and no reference to the map is retained, as illustrated  \* in the following code fragment:  \* <pre>  \* Set&lt;Object&gt; weakHashSet = Collections.newSetFromMap(  \* new WeakHashMap&lt;Object, Boolean&gt;());  \* </pre>  \*  \* @param <E> the class of the map keys and of the objects in the  \* returned set  \* @param map the backing map  \* @return the set backed by the map  \* @throws IllegalArgumentException if <tt>map</tt> is not empty  \* @since 1.6  \*/  public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) {  return new SetFromMap<>(map);  }  /\*\*  \* @serial include  \*/  private static class SetFromMap<E> extends AbstractSet<E>  implements Set<E>, Serializable  {  private final Map<E, Boolean> m; // The backing map  private transient Set<E> s; // Its keySet  SetFromMap(Map<E, Boolean> map) {  if (!map.isEmpty())  throw new IllegalArgumentException("Map is non-empty");  m = map;  s = map.keySet();  }  public void clear() { m.clear(); }  public int size() { return m.size(); }  public boolean isEmpty() { return m.isEmpty(); }  public boolean contains(Object o) { return m.containsKey(o); }  public boolean remove(Object o) { return m.remove(o) != null; }  public boolean add(E e) { return m.put(e, Boolean.TRUE) == null; }  public Iterator<E> iterator() { return s.iterator(); }  public Object[] toArray() { return s.toArray(); }  public <T> T[] toArray(T[] a) { return s.toArray(a); }  public String toString() { return s.toString(); }  public int hashCode() { return s.hashCode(); }  public boolean equals(Object o) { return o == this || s.equals(o); }  public boolean containsAll(Collection<?> c) {return s.containsAll(c);}  public boolean removeAll(Collection<?> c) {return s.removeAll(c);}  public boolean retainAll(Collection<?> c) {return s.retainAll(c);}  // addAll is the only inherited implementation  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {  s.forEach(action);  }  @Override  public boolean removeIf(Predicate<? super E> filter) {  return s.removeIf(filter);  }  @Override  public Spliterator<E> spliterator() {return s.spliterator();}  @Override  public Stream<E> stream() {return s.stream();}  @Override  public Stream<E> parallelStream() {return s.parallelStream();}  private static final long serialVersionUID = 2454657854757543876L;  private void readObject(java.io.ObjectInputStream stream)  throws IOException, ClassNotFoundException  {  stream.defaultReadObject();  s = m.keySet();  }  }  /\*\*  \* Returns a view of a {@link Deque} as a Last-in-first-out (Lifo)  \* {@link Queue}. Method <tt>add</tt> is mapped to <tt>push</tt>,  \* <tt>remove</tt> is mapped to <tt>pop</tt> and so on. This  \* view can be useful when you would like to use a method  \* requiring a <tt>Queue</tt> but you need Lifo ordering.  \*  \* <p>Each method invocation on the queue returned by this method  \* results in exactly one method invocation on the backing deque, with  \* one exception. The {@link Queue#addAll addAll} method is  \* implemented as a sequence of {@link Deque#addFirst addFirst}  \* invocations on the backing deque.  \*  \* @param <T> the class of the objects in the deque  \* @param deque the deque  \* @return the queue  \* @since 1.6  \*/  public static <T> Queue<T> asLifoQueue(Deque<T> deque) {  return new AsLIFOQueue<>(deque);  }  /\*\*  \* @serial include  \*/  static class AsLIFOQueue<E> extends AbstractQueue<E>  implements Queue<E>, Serializable {  private static final long serialVersionUID = 1802017725587941708L;  private final Deque<E> q;  AsLIFOQueue(Deque<E> q) { this.q = q; }  public boolean add(E e) { q.addFirst(e); return true; }  public boolean offer(E e) { return q.offerFirst(e); }  public E poll() { return q.pollFirst(); }  public E remove() { return q.removeFirst(); }  public E peek() { return q.peekFirst(); }  public E element() { return q.getFirst(); }  public void clear() { q.clear(); }  public int size() { return q.size(); }  public boolean isEmpty() { return q.isEmpty(); }  public boolean contains(Object o) { return q.contains(o); }  public boolean remove(Object o) { return q.remove(o); }  public Iterator<E> iterator() { return q.iterator(); }  public Object[] toArray() { return q.toArray(); }  public <T> T[] toArray(T[] a) { return q.toArray(a); }  public String toString() { return q.toString(); }  public boolean containsAll(Collection<?> c) {return q.containsAll(c);}  public boolean removeAll(Collection<?> c) {return q.removeAll(c);}  public boolean retainAll(Collection<?> c) {return q.retainAll(c);}  // We use inherited addAll; forwarding addAll would be wrong  // Override default methods in Collection  @Override  public void forEach(Consumer<? super E> action) {q.forEach(action);}  @Override  public boolean removeIf(Predicate<? super E> filter) {  return q.removeIf(filter);  }  @Override  public Spliterator<E> spliterator() {return q.spliterator();}  @Override  public Stream<E> stream() {return q.stream();}  @Override  public Stream<E> parallelStream() {return q.parallelStream();}  } |

## List (список)

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.util;  public interface List<E> extends Collection<E> {  ...  } |

Методы:

|  |  |
| --- | --- |
| 1. | int size() |

количество элементов в List (не вернет число больше Integer.MAX\_VALUE);

|  |  |
| --- | --- |
| 1. | boolean isEmpty() |

true, если List не содержит элементов;

|  |  |
| --- | --- |
| 1. | boolean contains(Object o) |

true, если List содержит указанный элемент;

|  |  |
| --- | --- |
| 1. | Iterator<E> iterator() |

вызов Iterator для обхода List;

|  |  |
| --- | --- |
| 1. | Object[] toArray() |
| 1. | <T> T[] toArray(T[] a) |

массив из элементов List (надо разобрать разницу!!!);

|  |  |
| --- | --- |
| 1. | boolean add(E e) |

добавление элемента в конец List;

|  |  |
| --- | --- |
| 1. | boolean remove(Object o) |

удаление элемента из List, если несколько — удалит первый;

|  |  |
| --- | --- |
| 1. | boolean containsAll(Collection<?> c); |

true, если List содержит все элементы указанной Collection;

|  |  |
| --- | --- |
| 1. | boolean addAll(Collection<? extends E> c) |

добавление элементов указанной Collection в конец List, порядок Collection определяется итератором;

|  |  |
| --- | --- |
| 1.  2.  3. | boolean addAll(  int index,  Collection<? extends E> c) |

вставка элементов указанной Collection в List с указанной позиции, порядок Collection определяется итератором;

|  |  |
| --- | --- |
| 1. | boolean removeAll(Collection<?> c); |

удаление элементов указанной Collection;

|  |  |
| --- | --- |
| 1. | boolean retainAll(Collection<?> c) |

сохранение элементов List и переданной Collection;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | default void replaceAll(  UnaryOperator<E> operator) {  Objects.requireNonNull(operator);  final ListIterator<E> li =  this.listIterator();  while (li.hasNext()) {  li.set(operator.apply(li.next()));  }  } |

изменение всех элементов коллекции с помощью UnaryOperator (T apply(T t));

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | default void sort(Comparator<? super E> c) {  Object[] a = this.toArray();  Arrays.sort(a, (Comparator) c);  ListIterator<E> i = this.listIterator();  for (Object e : a) {  i.next();  i.set((E) e);  }  } |

сортировка элементов в соответствии с Comparator;

|  |  |
| --- | --- |
| 1. | void clear() |

удаление всех элементов;

|  |  |
| --- | --- |
| 1. | boolean equals(Object o) |

сравнивает указанный объект с List;

|  |  |
| --- | --- |
| 1. | int hashCode() |

хеш‑код List;

|  |  |
| --- | --- |
| 1. | E get(int index) |

получить элемент по индексу;

|  |  |
| --- | --- |
| 1. | E set(int index, E element) |

заменить элемент по индексу;

|  |  |
| --- | --- |
| 1. | void add(int index, E element) |

вставить элемент по индексу;

|  |  |
| --- | --- |
| 1. | E remove(int index) |

удалить элемент по индексу;

|  |  |
| --- | --- |
| 1. | int indexOf(Object o) |

получить индекс указанного элемента, если элементов несколько — первый индекс, если нет таких элементов — «-1»;

|  |  |
| --- | --- |
| 1. | int lastIndexOf(Object o) |

indexOf(Object o), если элементов несколько — последний индекс;

|  |  |
| --- | --- |
| 1. | ListIterator<E> listIterator() |

вызов ListIterator для обхода List;

|  |  |
| --- | --- |
| 1. | ListIterator<E> listIterator(int index) |

вызов ListIterator для обхода List с указанной позиции;

|  |  |
| --- | --- |
| 1. | List<E> subList(int fromIndex, int toIndex) |

получение представления List между указанными индексами (последний не включительно), все изменения над полученным List будут применены к основному;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | @Override  default Spliterator<E> spliterator() {  return Spliterators.spliterator(  this,  Spliterator.ORDERED);  } |

Spliterator???.

## Set

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.util;  public interface Set<E> extends Collection<E> {  ...  } |

Методы:

|  |  |
| --- | --- |
| 1. | int size() |

количество элементов в Set (не вернет число больше Integer.MAX\_VALUE);

|  |  |
| --- | --- |
| 1. | boolean isEmpty() |

true, если Set не содержит элементов;

|  |  |
| --- | --- |
| 1. | boolean contains(Object o) |

true, если Set содержит указанный элемент;

|  |  |
| --- | --- |
| 1. | Iterator<E> iterator() |

вызов Iterator для обхода Set;

|  |  |
| --- | --- |
| 1. | Object[] toArray() |
| 1. | <T> T[] toArray(T[] a) |

массив из элементов Set (надо разобрать разницу!!!);

|  |  |
| --- | --- |
| 1. | boolean add(E e) |

добавление элемента в Set, если Set не содержит такого элемента;

|  |  |
| --- | --- |
| 1. | boolean remove(Object o) |

удаление элемента из Set;

|  |  |
| --- | --- |
| 1. | boolean containsAll(Collection<?> c); |

true, если Set содержит все элементы указанной Collection;

|  |  |
| --- | --- |
| 1. | boolean addAll(Collection<? extends E> c) |

добавление элементов указанной Collection в Set, если они еще не присутствуют;

|  |  |
| --- | --- |
| 1. | boolean retainAll(Collection<?> c) |

сохранение только тех элементов, которые присутствуют и в Set, и в Collection;

|  |  |
| --- | --- |
| 1. | boolean removeAll(Collection<?> c); |

удаление элементов указанной Collection;

|  |  |
| --- | --- |
| 1. | void clear() |

удаление всех элементов;

|  |  |
| --- | --- |
| 1. | boolean equals(Object o) |

сравнивает указанный объект с Set;

|  |  |
| --- | --- |
| 1. | int hashCode() |

хеш‑код List;

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | @Override  default Spliterator<E> spliterator() {  return Spliterators.spliterator(  this,  Spliterator.DISTINCT);  } |

Spliterator???.

## Queue

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.util;  public interface Queue<E> extends Collection<E> {  ...  } |

Методы:

|  |  |
| --- | --- |
| 1. | boolean add(E e) |

sdfsdf

|  |  |
| --- | --- |
| 1. | boolean offer(E e) |

sdfsdf

|  |  |
| --- | --- |
| 1. | E remove() |

sdfsdf

|  |  |
| --- | --- |
| 1. | E poll() |

sdfsdf

|  |  |
| --- | --- |
| 1. | E element() |

sdfsdf

|  |  |
| --- | --- |
| 1. | E peek() |

sdfsdf