package java.util;

import java.util.function.BiConsumer;

import java.util.function.BiFunction;

import java.util.function.Function;

import java.io.Serializable;

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

\* @param <K> the type of keys maintained by this map

\* @param <V> the type of mapped values

**public** **interface** Map<K,V> {

// Query Operations

\* **@return** the number of key-value mappings in this map

**int** size();

\* **@return** <tt>true</tt> if this map contains no key-value mappings

**boolean** isEmpty();

/\*\*

\* Returns <tt>true</tt> if this map contains a mapping for the specified

\* key

\* **@throws** ClassCastException if the key is of an inappropriate type for

\* this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key is null and this map

\* does not permit null keys

\*/

**boolean** containsKey(Object key);

/\*\*

\* Returns <tt>true</tt> if this map maps one or more keys to the

\* specified value.

\*/

**boolean** containsValue(Object value);

\* **@throws** NullPointerException if the specified key is null and this map

\* does not permit null keys

\*/

V get(Object key);

// Modification Operations

V put(K key, V value);

V remove(Object key);

// Bulk Operations

/\*\*

\* Copies all of the mappings from the specified map to this map

\* (optional operation). The effect of this call is equivalent to that

\* of calling {@link #put(Object,Object) put(k, v)} on this map once

\* for each mapping from key <tt>k</tt> to value <tt>v</tt> in the

\* specified map. The behavior of this operation is undefined if the

\* specified map is modified while the operation is in progress.

\*

\* **@param** m mappings to be stored in this map

\* **@throws** UnsupportedOperationException if the <tt>putAll</tt> operation

\* is not supported by this map

\* **@throws** ClassCastException if the class of a key or value in the

\* specified map prevents it from being stored in this map

\* **@throws** NullPointerException if the specified map is null, or if

\* this map does not permit null keys or values, and the

\* specified map contains null keys or values

\* **@throws** IllegalArgumentException if some property of a key or value in

\* the specified map prevents it from being stored in this map

\*/

**void** putAll(Map<? **extends** K, ? **extends** V> m);

/\*\*

\* Removes all of the mappings from this map (optional operation).

\* The map will be empty after this call returns.

\*

\* **@throws** UnsupportedOperationException if the <tt>clear</tt> operation

\* is not supported by this map

\*/

**void** clear();

// Views

/\*\*

\* Returns a {@link Set} view of the keys contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation), the results of

\* the iteration are undefined. The set supports element removal,

\* which removes the corresponding mapping from the map, via the

\* <tt>Iterator.remove</tt>, <tt>Set.remove</tt>,

\* <tt>removeAll</tt>, <tt>retainAll</tt>, and <tt>clear</tt>

\* operations. It does not support the <tt>add</tt> or <tt>addAll</tt>

\* operations.

\*

\* **@return** a set view of the keys contained in this map

\*/

Set<K> keySet();

/\*\*

\* Returns a {@link Collection} view of the values contained in this map.

\* The collection is backed by the map, so changes to the map are

\* reflected in the collection, and vice-versa. If the map is

\* modified while an iteration over the collection is in progress

\* (except through the iterator's own <tt>remove</tt> operation),

\* the results of the iteration are undefined. The collection

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Collection.remove</tt>, <tt>removeAll</tt>,

\* <tt>retainAll</tt> and <tt>clear</tt> operations. It does not

\* support the <tt>add</tt> or <tt>addAll</tt> operations.

\*

\* **@return** a collection view of the values contained in this map

\*/

Collection<V> values();

/\*\*

\* Returns a {@link Set} view of the mappings contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation, or through the

\* <tt>setValue</tt> operation on a map entry returned by the

\* iterator) the results of the iteration are undefined. The set

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Set.remove</tt>, <tt>removeAll</tt>, <tt>retainAll</tt> and

\* <tt>clear</tt> operations. It does not support the

\* <tt>add</tt> or <tt>addAll</tt> operations.

\*

\* **@return** a set view of the mappings contained in this map

\*/

Set<Map.Entry<K, V>> entrySet();

/\*\*

\* A map entry (key-value pair). The <tt>Map.entrySet</tt> method returns

\* a collection-view of the map, whose elements are of this class. The

\* <i>only</i> way to obtain a reference to a map entry is from the

\* iterator of this collection-view.

These <tt>Map.Entry</tt> objects are

\* valid <i>only</i> for the duration of the iteration; more formally,

\* the behavior of a map entry is undefined if the backing map has been

\* modified after the entry was returned by the iterator, except through

\* the <tt>setValue</tt> operation on the map entry.

\*

\* **@see** Map#entrySet()

\* **@since** 1.2

\*/

**interface** Entry<K,V> {

/\*\*

\* Returns the key corresponding to this entry.

K getKey();

/\*\*

\* Returns the value corresponding to this entry. If the mapping

\* has been removed from the backing map (by the iterator's

\* <tt>remove</tt> operation), the results of this call are undefined.

\*

\* **@return** the value corresponding to this entry

V getValue();

/\*\*

\* Replaces the value corresponding to this entry with the specified

\* value (optional operation). (Writes through to the map.) The

\* behavior of this call is undefined if the mapping has already been

\* removed from the map (by the iterator's <tt>remove</tt> operation).

\*

\* **@param** value new value to be stored in this entry

\* **@return** old value corresponding to the entry

\* **@throws** UnsupportedOperationException if the <tt>put</tt> operation

\* is not supported by the backing map

\* **@throws** ClassCastException if the class of the specified value

\* prevents it from being stored in the backing map

\* **@throws** NullPointerException if the backing map does not permit

\* null values, and the specified value is null

\* **@throws** IllegalArgumentException if some property of this value

\* prevents it from being stored in the backing map

\* **@throws** IllegalStateException implementations may, but are not

\* required to, throw this exception if the entry has been

\* removed from the backing map.

\*/

V setValue(V value);

/\*\*

\* Compares the specified object with this entry for equality.

\* Returns <tt>true</tt> if the given object is also a map entry and

\* the two entries represent the same mapping. More formally, two

\* entries <tt>e1</tt> and <tt>e2</tt> represent the same mapping

\* if<pre>

\* (e1.getKey()==null ?

\* e2.getKey()==null : e1.getKey().equals(e2.getKey())) &amp;&amp;

\* (e1.getValue()==null ?

\* e2.getValue()==null : e1.getValue().equals(e2.getValue()))

\* </pre>

\* This ensures that the <tt>equals</tt> method works properly across

\* different implementations of the <tt>Map.Entry</tt> interface.

\*

\* **@param** o object to be compared for equality with this map entry

\* **@return** <tt>true</tt> if the specified object is equal to this map

\* entry

\*/

**boolean** equals(Object o);

/\*\*

\* Returns the hash code value for this map entry. The hash code

\* of a map entry <tt>e</tt> is defined to be: <pre>

\* (e.getKey()==null ? 0 : e.getKey().hashCode()) ^

\* (e.getValue()==null ? 0 : e.getValue().hashCode())

\* </pre>

\* This ensures that <tt>e1.equals(e2)</tt> implies that

\* <tt>e1.hashCode()==e2.hashCode()</tt> for any two Entries

\* <tt>e1</tt> and <tt>e2</tt>, as required by the general

\* contract of <tt>Object.hashCode</tt>.

\*

\* **@return** the hash code value for this map entry

\* **@see** Object#hashCode()

\* **@see** Object#equals(Object)

\* **@see** #equals(Object)

\*/

**int** hashCode();

/\*\*

\* Returns a comparator that compares {@link Map.Entry} in natural order on key.

\*

\* <p>The returned comparator is serializable and throws {@link

\* NullPointerException} when comparing an entry with a null key.

\*

\* **@param** <K> the {@link Comparable} type of then map keys

\* **@param** <V> the type of the map values

\* **@return** a comparator that compares {@link Map.Entry} in natural order on key.

\* **@see** Comparable

\* **@since** 1.8

\*/

**public** **static** <K **extends** Comparable<? **super** K>, V> Comparator<Map.Entry<K,V>> comparingByKey() {

**return** (Comparator<Map.Entry<K, V>> & Serializable)

(c1, c2) -> c1.getKey().compareTo(c2.getKey());

}

/\*\*

\* Returns a comparator that compares {@link Map.Entry} in natural order on value.

\*

\* <p>The returned comparator is serializable and throws {@link

\* NullPointerException} when comparing an entry with null values.

\*

\* **@param** <K> the type of the map keys

\* **@param** <V> the {@link Comparable} type of the map values

\* **@return** a comparator that compares {@link Map.Entry} in natural order on value.

\* **@see** Comparable

\* **@since** 1.8

\*/

**public** **static** <K, V **extends** Comparable<? **super** V>> Comparator<Map.Entry<K,V>> comparingByValue() {

**return** (Comparator<Map.Entry<K, V>> & Serializable)

(c1, c2) -> c1.getValue().compareTo(c2.getValue());

}

/\*\*

\* Returns a comparator that compares {@link Map.Entry} by key using the given

\* {@link Comparator}.

\*

\* <p>The returned comparator is serializable if the specified comparator

\* is also serializable.

\*

\* **@param** <K> the type of the map keys

\* **@param** <V> the type of the map values

\* **@param** cmp the key {@link Comparator}

\* **@return** a comparator that compares {@link Map.Entry} by the key.

\* **@since** 1.8

\*/

**public** **static** <K, V> Comparator<Map.Entry<K, V>> comparingByKey(Comparator<? **super** K> cmp) {

Objects.*requireNonNull*(cmp);

**return** (Comparator<Map.Entry<K, V>> & Serializable)

(c1, c2) -> cmp.compare(c1.getKey(), c2.getKey());

}

/\*\*

\* Returns a comparator that compares {@link Map.Entry} by value using the given

\* {@link Comparator}.

\*

\* <p>The returned comparator is serializable if the specified comparator

\* is also serializable.

\*

\* **@param** <K> the type of the map keys

\* **@param** <V> the type of the map values

\* **@param** cmp the value {@link Comparator}

\* **@return** a comparator that compares {@link Map.Entry} by the value.

\* **@since** 1.8

\*/

**public** **static** <K, V> Comparator<Map.Entry<K, V>> comparingByValue(Comparator<? **super** V> cmp) {

Objects.*requireNonNull*(cmp);

**return** (Comparator<Map.Entry<K, V>> & Serializable)

(c1, c2) -> cmp.compare(c1.getValue(), c2.getValue());

}

}

// Comparison and hashing

/\*\*

\* Compares the specified object with this map for equality. Returns

\* <tt>true</tt> if the given object is also a map and the two maps

\* represent the same mappings. More formally, two maps <tt>m1</tt> and

\* <tt>m2</tt> represent the same mappings if

\* <tt>m1.entrySet().equals(m2.entrySet())</tt>. This ensures that the

\* <tt>equals</tt> method works properly across different implementations

\* of the <tt>Map</tt> interface.

\*

\* **@param** o object to be compared for equality with this map

\* **@return** <tt>true</tt> if the specified object is equal to this map

\*/

**boolean** equals(Object o);

/\*\*

\* Returns the hash code value for this map. The hash code of a map is

\* defined to be the sum of the hash codes of each entry in the map's

\* <tt>entrySet()</tt> view. This ensures that <tt>m1.equals(m2)</tt>

\* implies that <tt>m1.hashCode()==m2.hashCode()</tt> for any two maps

\* <tt>m1</tt> and <tt>m2</tt>, as required by the general contract of

\* {@link Object#hashCode}.

\*

\* **@return** the hash code value for this map

\* **@see** Map.Entry#hashCode()

\* **@see** Object#equals(Object)

\* **@see** #equals(Object)

\*/

**int** hashCode();

// Defaultable methods

/\*\*

\* Returns the value to which the specified key is mapped, or

\* {@code defaultValue} if this map contains no mapping for the key.

\*

\* **@implSpec**

\* The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** key the key whose associated value is to be returned

\* **@param** defaultValue the default mapping of the key

\* **@return** the value to which the specified key is mapped, or

\* {@code defaultValue} if this map contains no mapping for the key

\* **@throws** ClassCastException if the key is of an inappropriate type for

\* this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key is null and this map

\* does not permit null keys

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** V getOrDefault(Object key, V defaultValue) {

V v;

**return** (((v = get(key)) != **null**) || containsKey(key))

? v

: defaultValue;

}

/\*\*

\* Performs the given action for each entry in this map until all entries

\* have been processed or the action throws an exception. Unless

\* otherwise specified by the implementing class, actions are performed in

\* the order of entry set iteration (if an iteration order is specified.)

\* Exceptions thrown by the action are relayed to the caller.

\*

\* **@implSpec**

\* The default implementation is equivalent to, for this {@code map}:

\* <pre> {@code

\* for (Map.Entry<K, V> entry : map.entrySet())

\* action.accept(entry.getKey(), entry.getValue());

\* }</pre>

\*

\* The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** action The action to be performed for each entry

\* **@throws** NullPointerException if the specified action is null

\* **@throws** ConcurrentModificationException if an entry is found to be

\* removed during iteration

\* **@since** 1.8

\*/

**default** **void** forEach(BiConsumer<? **super** K, ? **super** V> action) {

Objects.*requireNonNull*(action);

**for** (Map.Entry<K, V> entry : entrySet()) {

K k;

V v;

**try** {

k = entry.getKey();

v = entry.getValue();

} **catch**(IllegalStateException ise) {

// this usually means the entry is no longer in the map.

**throw** **new** ConcurrentModificationException(ise);

}

action.accept(k, v);

}

}

/\*\*

\* Replaces each entry's value with the result of invoking the given

\* function on that entry until all entries have been processed or the

\* function throws an exception. Exceptions thrown by the function are

\* relayed to the caller.

\*

\* **@implSpec**

\* <p>The default implementation is equivalent to, for this {@code map}:

\* <pre> {@code

\* for (Map.Entry<K, V> entry : map.entrySet())

\* entry.setValue(function.apply(entry.getKey(), entry.getValue()));

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** function the function to apply to each entry

\* **@throws** UnsupportedOperationException if the {@code set} operation

\* is not supported by this map's entry set iterator.

\* **@throws** ClassCastException if the class of a replacement value

\* prevents it from being stored in this map

\* **@throws** NullPointerException if the specified function is null, or the

\* specified replacement value is null, and this map does not permit null

\* values

\* **@throws** ClassCastException if a replacement value is of an inappropriate

\* type for this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if function or a replacement value is null,

\* and this map does not permit null keys or values

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** IllegalArgumentException if some property of a replacement value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ConcurrentModificationException if an entry is found to be

\* removed during iteration

\* **@since** 1.8

\*/

**default** **void** replaceAll(BiFunction<? **super** K, ? **super** V, ? **extends** V> function) {

Objects.*requireNonNull*(function);

**for** (Map.Entry<K, V> entry : entrySet()) {

K k;

V v;

**try** {

k = entry.getKey();

v = entry.getValue();

} **catch**(IllegalStateException ise) {

// this usually means the entry is no longer in the map.

**throw** **new** ConcurrentModificationException(ise);

}

// ise thrown from function is not a cme.

v = function.apply(k, v);

**try** {

entry.setValue(v);

} **catch**(IllegalStateException ise) {

// this usually means the entry is no longer in the map.

**throw** **new** ConcurrentModificationException(ise);

}

}

}

/\*\*

\* If the specified key is not already associated with a value (or is mapped

\* to {@code null}) associates it with the given value and returns

\* {@code null}, else returns the current value.

\*

\* **@implSpec**

\* The default implementation is equivalent to, for this {@code

\* map}:

\*

\* <pre> {@code

\* V v = map.get(key);

\* if (v == null)

\* v = map.put(key, value);

\*

\* return v;

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** key key with which the specified value is to be associated

\* **@param** value value to be associated with the specified key

\* **@return** the previous value associated with the specified key, or

\* {@code null} if there was no mapping for the key.

\* (A {@code null} return can also indicate that the map

\* previously associated {@code null} with the key,

\* if the implementation supports null values.)

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the key or value is of an inappropriate

\* type for this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key or value is null,

\* and this map does not permit null keys or values

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** IllegalArgumentException if some property of the specified key

\* or value prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** V putIfAbsent(K key, V value) {

V v = get(key);

**if** (v == **null**) {

v = put(key, value);

}

**return** v;

}

/\*\*

\* Removes the entry for the specified key only if it is currently

\* mapped to the specified value.

\*

\* **@implSpec**

\* The default implementation is equivalent to, for this {@code map}:

\*

\* <pre> {@code

\* if (map.containsKey(key) && Objects.equals(map.get(key), value)) {

\* map.remove(key);

\* return true;

\* } else

\* return false;

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** key key with which the specified value is associated

\* **@param** value value expected to be associated with the specified key

\* **@return** {@code true} if the value was removed

\* **@throws** UnsupportedOperationException if the {@code remove} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the key or value is of an inappropriate

\* type for this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key or value is null,

\* and this map does not permit null keys or values

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** **boolean** remove(Object key, Object value) {

Object curValue = get(key);

**if** (!Objects.*equals*(curValue, value) ||

(curValue == **null** && !containsKey(key))) {

**return** **false**;

}

remove(key);

**return** **true**;

}

/\*\*

\* Replaces the entry for the specified key only if currently

\* mapped to the specified value.

\*

\* **@implSpec**

\* The default implementation is equivalent to, for this {@code map}:

\*

\* <pre> {@code

\* if (map.containsKey(key) && Objects.equals(map.get(key), value)) {

\* map.put(key, newValue);

\* return true;

\* } else

\* return false;

\* }</pre>

\*

\* The default implementation does not throw NullPointerException

\* for maps that do not support null values if oldValue is null unless

\* newValue is also null.

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** key key with which the specified value is associated

\* **@param** oldValue value expected to be associated with the specified key

\* **@param** newValue value to be associated with the specified key

\* **@return** {@code true} if the value was replaced

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of a specified key or value

\* prevents it from being stored in this map

\* **@throws** NullPointerException if a specified key or newValue is null,

\* and this map does not permit null keys or values

\* **@throws** NullPointerException if oldValue is null and this map does not

\* permit null values

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** IllegalArgumentException if some property of a specified key

\* or value prevents it from being stored in this map

\* **@since** 1.8

\*/

**default** **boolean** replace(K key, V oldValue, V newValue) {

Object curValue = get(key);

**if** (!Objects.*equals*(curValue, oldValue) ||

(curValue == **null** && !containsKey(key))) {

**return** **false**;

}

put(key, newValue);

**return** **true**;

}

/\*\*

\* Replaces the entry for the specified key only if it is

\* currently mapped to some value.

\*

\* **@implSpec**

\* The default implementation is equivalent to, for this {@code map}:

\*

\* <pre> {@code

\* if (map.containsKey(key)) {

\* return map.put(key, value);

\* } else

\* return null;

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties.

\*

\* **@param** key key with which the specified value is associated

\* **@param** value value to be associated with the specified key

\* **@return** the previous value associated with the specified key, or

\* {@code null} if there was no mapping for the key.

\* (A {@code null} return can also indicate that the map

\* previously associated {@code null} with the key,

\* if the implementation supports null values.)

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of the specified key or value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key or value is null,

\* and this map does not permit null keys or values

\* **@throws** IllegalArgumentException if some property of the specified key

\* or value prevents it from being stored in this map

\* **@since** 1.8

\*/

**default** V replace(K key, V value) {

V curValue;

**if** (((curValue = get(key)) != **null**) || containsKey(key)) {

curValue = put(key, value);

}

**return** curValue;

}

/\*\*

\* If the specified key is not already associated with a value (or is mapped

\* to {@code null}), attempts to compute its value using the given mapping

\* function and enters it into this map unless {@code null}.

\*

\* <p>If the function returns {@code null} no mapping is recorded. If

\* the function itself throws an (unchecked) exception, the

\* exception is rethrown, and no mapping is recorded. The most

\* common usage is to construct a new object serving as an initial

\* mapped value or memoized result, as in:

\*

\* <pre> {@code

\* map.computeIfAbsent(key, k -> new Value(f(k)));

\* }</pre>

\*

\* <p>Or to implement a multi-value map, {@code Map<K,Collection<V>>},

\* supporting multiple values per key:

\*

\* <pre> {@code

\* map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);

\* }</pre>

\*

\*

\* **@implSpec**

\* The default implementation is equivalent to the following steps for this

\* {@code map}, then returning the current value or {@code null} if now

\* absent:

\*

\* <pre> {@code

\* if (map.get(key) == null) {

\* V newValue = mappingFunction.apply(key);

\* if (newValue != null)

\* map.put(key, newValue);

\* }

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties. In particular, all implementations of

\* subinterface {@link java.util.concurrent.ConcurrentMap} must document

\* whether the function is applied once atomically only if the value is not

\* present.

\*

\* **@param** key key with which the specified value is to be associated

\* **@param** mappingFunction the function to compute a value

\* **@return** the current (existing or computed) value associated with

\* the specified key, or null if the computed value is null

\* **@throws** NullPointerException if the specified key is null and

\* this map does not support null keys, or the mappingFunction

\* is null

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of the specified key or value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** V computeIfAbsent(K key,

Function<? **super** K, ? **extends** V> mappingFunction) {

Objects.*requireNonNull*(mappingFunction);

V v;

**if** ((v = get(key)) == **null**) {

V newValue;

**if** ((newValue = mappingFunction.apply(key)) != **null**) {

put(key, newValue);

**return** newValue;

}

}

**return** v;

}

/\*\*

\* If the value for the specified key is present and non-null, attempts to

\* compute a new mapping given the key and its current mapped value.

\*

\* <p>If the function returns {@code null}, the mapping is removed. If the

\* function itself throws an (unchecked) exception, the exception is

\* rethrown, and the current mapping is left unchanged.

\*

\* **@implSpec**

\* The default implementation is equivalent to performing the following

\* steps for this {@code map}, then returning the current value or

\* {@code null} if now absent:

\*

\* <pre> {@code

\* if (map.get(key) != null) {

\* V oldValue = map.get(key);

\* V newValue = remappingFunction.apply(key, oldValue);

\* if (newValue != null)

\* map.put(key, newValue);

\* else

\* map.remove(key);

\* }

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties. In particular, all implementations of

\* subinterface {@link java.util.concurrent.ConcurrentMap} must document

\* whether the function is applied once atomically only if the value is not

\* present.

\*

\* **@param** key key with which the specified value is to be associated

\* **@param** remappingFunction the function to compute a value

\* **@return** the new value associated with the specified key, or null if none

\* **@throws** NullPointerException if the specified key is null and

\* this map does not support null keys, or the

\* remappingFunction is null

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of the specified key or value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** V computeIfPresent(K key,

BiFunction<? **super** K, ? **super** V, ? **extends** V> remappingFunction) {

Objects.*requireNonNull*(remappingFunction);

V oldValue;

**if** ((oldValue = get(key)) != **null**) {

V newValue = remappingFunction.apply(key, oldValue);

**if** (newValue != **null**) {

put(key, newValue);

**return** newValue;

} **else** {

remove(key);

**return** **null**;

}

} **else** {

**return** **null**;

}

}

/\*\*

\* Attempts to compute a mapping for the specified key and its current

\* mapped value (or {@code null} if there is no current mapping). For

\* example, to either create or append a {@code String} msg to a value

\* mapping:

\*

\* <pre> {@code

\* map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))}</pre>

\* (Method {@link #merge merge()} is often simpler to use for such purposes.)

\*

\* <p>If the function returns {@code null}, the mapping is removed (or

\* remains absent if initially absent). If the function itself throws an

\* (unchecked) exception, the exception is rethrown, and the current mapping

\* is left unchanged.

\*

\* **@implSpec**

\* The default implementation is equivalent to performing the following

\* steps for this {@code map}, then returning the current value or

\* {@code null} if absent:

\*

\* <pre> {@code

\* V oldValue = map.get(key);

\* V newValue = remappingFunction.apply(key, oldValue);

\* if (oldValue != null ) {

\* if (newValue != null)

\* map.put(key, newValue);

\* else

\* map.remove(key);

\* } else {

\* if (newValue != null)

\* map.put(key, newValue);

\* else

\* return null;

\* }

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties. In particular, all implementations of

\* subinterface {@link java.util.concurrent.ConcurrentMap} must document

\* whether the function is applied once atomically only if the value is not

\* present.

\*

\* **@param** key key with which the specified value is to be associated

\* **@param** remappingFunction the function to compute a value

\* **@return** the new value associated with the specified key, or null if none

\* **@throws** NullPointerException if the specified key is null and

\* this map does not support null keys, or the

\* remappingFunction is null

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of the specified key or value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@since** 1.8

\*/

**default** V compute(K key,

BiFunction<? **super** K, ? **super** V, ? **extends** V> remappingFunction) {

Objects.*requireNonNull*(remappingFunction);

V oldValue = get(key);

V newValue = remappingFunction.apply(key, oldValue);

**if** (newValue == **null**) {

// delete mapping

**if** (oldValue != **null** || containsKey(key)) {

// something to remove

remove(key);

**return** **null**;

} **else** {

// nothing to do. Leave things as they were.

**return** **null**;

}

} **else** {

// add or replace old mapping

put(key, newValue);

**return** newValue;

}

}

/\*\*

\* If the specified key is not already associated with a value or is

\* associated with null, associates it with the given non-null value.

\* Otherwise, replaces the associated value with the results of the given

\* remapping function, or removes if the result is {@code null}. This

\* method may be of use when combining multiple mapped values for a key.

\* For example, to either create or append a {@code String msg} to a

\* value mapping:

\*

\* <pre> {@code

\* map.merge(key, msg, String::concat)

\* }</pre>

\*

\* <p>If the function returns {@code null} the mapping is removed. If the

\* function itself throws an (unchecked) exception, the exception is

\* rethrown, and the current mapping is left unchanged.

\*

\* **@implSpec**

\* The default implementation is equivalent to performing the following

\* steps for this {@code map}, then returning the current value or

\* {@code null} if absent:

\*

\* <pre> {@code

\* V oldValue = map.get(key);

\* V newValue = (oldValue == null) ? value :

\* remappingFunction.apply(oldValue, value);

\* if (newValue == null)

\* map.remove(key);

\* else

\* map.put(key, newValue);

\* }</pre>

\*

\* <p>The default implementation makes no guarantees about synchronization

\* or atomicity properties of this method. Any implementation providing

\* atomicity guarantees must override this method and document its

\* concurrency properties. In particular, all implementations of

\* subinterface {@link java.util.concurrent.ConcurrentMap} must document

\* whether the function is applied once atomically only if the value is not

\* present.

\*

\* **@param** key key with which the resulting value is to be associated

\* **@param** value the non-null value to be merged with the existing value

\* associated with the key or, if no existing value or a null value

\* is associated with the key, to be associated with the key

\* **@param** remappingFunction the function to recompute a value if present

\* **@return** the new value associated with the specified key, or null if no

\* value is associated with the key

\* **@throws** UnsupportedOperationException if the {@code put} operation

\* is not supported by this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** ClassCastException if the class of the specified key or value

\* prevents it from being stored in this map

\* (<a href="{@docRoot}/java/util/Collection.html#optional-restrictions">optional</a>)

\* **@throws** NullPointerException if the specified key is null and this map

\* does not support null keys or the value or remappingFunction is

\* null

\* **@since** 1.8

\*/

**default** V merge(K key, V value,

BiFunction<? **super** V, ? **super** V, ? **extends** V> remappingFunction) {

Objects.*requireNonNull*(remappingFunction);

Objects.*requireNonNull*(value);

V oldValue = get(key);

V newValue = (oldValue == **null**) ? value :

remappingFunction.apply(oldValue, value);

**if**(newValue == **null**) {

remove(key);

} **else** {

put(key, newValue);

}

**return** newValue;

}

}