compact1, compact2, compact3
java.util

Interface Map<K,V>

Type Parameters:

`K` - the type of keys maintained by this map

`V` - the type of mapped values

All Known Subinterfaces:

`Bindings`, `ConcurrentMap<K,V>`, `ConcurrentNavigableMap<K,V>`, `LogicalMessageContext`, `MessageContext`, `NavigableMap<K,V>`, `SOAPMessageContext`, `SortedMap<K,V>`

All Known Implementing Classes:

`AbstractMap`, `Attributes`, `AuthProvider`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `EnumMap`, `HashMap`, `Hashtable`, `IdentityHashMap`, `LinkedHashMap`, `PrinterStateReasons`, `Properties`, `Provider`, `RenderingHints`, `SimpleBindings`, `TabularDataSupport`, `TreeMap`, `UIDefaults`, `WeakHashMap`

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

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

This interface takes the place of the `Dictionary` class, which was a totally abstract class rather than an interface.

The `Map` interface provides three *collection views*, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The *order* of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the `TreeMap` class, make specific guarantees as to their order; others, like the `HashMap` class, do not.

Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.

All general-purpose map implementation classes should provide two "standard" constructors: a void (no arguments) constructor which creates an empty map, and a constructor with a single argument of type `Map`, which creates a new map with the same key-value mappings as its argument. In effect, the latter constructor allows the user to copy any map, producing an equivalent map of the desired class. There is no way to enforce this recommendation (as interfaces cannot contain constructors) but all of the general-purpose map implementations in the JDK comply.

The "destructive" methods contained in this interface, that is, the methods that modify the map on which they operate, are specified to throw `UnsupportedOperationException` if this map does not support the operation. If this is the case, these methods may, but are not required to, throw an `UnsupportedOperationException` if the invocation would have no effect on the map. For example, invoking the `putAll(Map)` method on an unmodifiable map may, but is not required to, throw the exception if the map whose mappings are to be "superimposed" is empty.

Some map implementations have restrictions on the keys and values they may contain. For example, some implementations prohibit null keys and values, and some have restrictions on the types of their keys. Attempting to insert an ineligible key or value throws an unchecked exception, typically `NullPointerException` or `ClassCastException`. Attempting to query the presence of an ineligible key or value may throw an exception, or it may simply return false; some implementations will exhibit the former behavior and some will exhibit the latter. More generally, attempting an operation on an ineligible key or value whose completion would not result in the insertion of an ineligible element into the map may throw an exception or it may succeed, at the option of the implementation. Such exceptions are marked as "optional" in the specification for this interface.

Many methods in Collections Framework interfaces are defined in terms of the `equals` method. For example, the specification for the `containsKey(Object key)` method says: "returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`." This specification should *not* be construed to imply that invoking `Map.containsKey` with a non-null argument `key` will cause `key.equals(k)` to be invoked for any key `k`. Implementations are free to implement optimizations whereby the `equals` invocation is avoided, for example, by first comparing the hash codes of the two keys. (The `Object.hashCode()` specification guarantees that two objects with unequal hash codes cannot be equal.) More generally, implementations of the various Collections Framework interfaces are free to take advantage of the specified behavior of underlying `Object` methods wherever the implementor deems it appropriate.

Some map operations which perform recursive traversal of the map may fail with an exception for self-referential instances where the map directly or indirectly contains itself. This includes the `clone()`, `equals()`, `hashCode()` and `toString()` methods. Implementations may optionally handle the self-referential scenario, however most current implementations do not do so.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`HashMap`, `TreeMap`, `Hashtable`, `SortedMap`, `Collection`, `Set`

Nested Class Summary

Nested Classes

Modifier and Type	Interface and Description
static interface	**`Map.Entry<K,V>`** A map entry (key-value pair).

Method Summary

All Methods Instance Methods Abstract Methods Default Methods

Modifier and Type	Method and Description
void	**`clear()`** Removes all of the mappings from this map (optional operation).
default V	**`compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)`** Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
default V	**`computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)`**

	If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.
default **V**	**computeIfPresent**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) 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.
boolean	**containsKey**(**Object** key) Returns `true` if this map contains a mapping for the specified key.
boolean	**containsValue**(**Object** value) Returns `true` if this map maps one or more keys to the specified value.
Set<**Map.Entry**<**K**,**V**>>	**entrySet**() Returns a **Set** view of the mappings contained in this map.
boolean	**equals**(**Object** o) Compares the specified object with this map for equality.
default void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
default **V**	**getOrDefault**(**Object** key, **V** defaultValue) Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this map.
boolean	**isEmpty**() Returns `true` if this map contains no key-value mappings.
Set<**K**>	**keySet**() Returns a **Set** view of the keys contained in this map.
default **V**	**merge**(**K** key, **V** value, **BiFunction**<? super **V**,? super **V**,? extends **V**> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map (optional operation).
void	**putAll**(**Map**<? extends **K**,? extends **V**> m) Copies all of the mappings from the specified map to this map (optional operation).
default **V**	**putIfAbsent**(**K** key, **V** value) If the specified key is not already associated with a value (or is mapped to `null`) associates it with the given value and returns `null`, else returns the current value.
V	**remove**(**Object** key)

	Removes the mapping for a key from this map if it is present (optional operation).
default boolean	**remove**(**Object** key, **Object** value) Removes the entry for the specified key only if it is currently mapped to the specified value.
default **V**	**replace**(**K** key, **V** value) Replaces the entry for the specified key only if it is currently mapped to some value.
default boolean	**replace**(**K** key, **V** oldValue, **V** newValue) Replaces the entry for the specified key only if currently mapped to the specified value.
default void	**replaceAll**(**BiFunction**<? super **K**,? super **V**,? extends **V**> function) 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.
int	**size**() Returns the number of key-value mappings in this map.
Collection<**V**>	**values**() Returns a **Collection** view of the values contained in this map.

Method Detail

size

```
int size()
```

Returns the number of key-value mappings in this map. If the map contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Returns:

the number of key-value mappings in this map

isEmpty

```
boolean isEmpty()
```

Returns `true` if this map contains no key-value mappings.

Returns:

true if this map contains no key-value mappings

containsKey

```
boolean containsKey(Object key)
```

Returns `true` if this map contains a mapping for the specified key. More formally, returns `true` if and only if this map contains a mapping for a key `k` such that `(key==null ? k==null : key.equals(k))`. (There can be at most one such mapping.)

Parameters:

key - key whose presence in this map is to be tested

Returns:

true if this map contains a mapping for the specified key

Throws:

ClassCastException - if the key is of an inappropriate type for this map (optional)

NullPointerException - if the specified key is null and this map does not permit null keys (optional)

containsValue

boolean containsValue(Object value)

Returns true if this map maps one or more keys to the specified value. More formally, returns true if and only if this map contains at least one mapping to a value v such that (value==null ? v==null : value.equals(v)). This operation will probably require time linear in the map size for most implementations of the Map interface.

Parameters:

value - value whose presence in this map is to be tested

Returns:

true if this map maps one or more keys to the specified value

Throws:

ClassCastException - if the value is of an inappropriate type for this map (optional)

NullPointerException - if the specified value is null and this map does not permit null values (optional)

get

V get(Object key)

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

More formally, if this map contains a mapping from a key k to a value v such that (key==null ? k==null : key.equals(k)), then this method returns v; otherwise it returns null. (There can be at most one such mapping.)

If this map permits null values, then a return value of null does not *necessarily* indicate that the map contains no mapping for the key; it's also possible that the map explicitly maps the key to null. The containsKey operation may be used to distinguish these two cases.

Parameters:

key - the key whose associated value is to be returned

Returns:

the value to which the specified key is mapped, or null if this map contains no mapping for the key

Throws:

ClassCastException - if the key is of an inappropriate type for this map (optional)

NullPointerException - if the specified key is null and this map does not permit null keys (optional)

put

```
V put(K key,
      V value)
```

Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for the key, the old value is replaced by the specified value. (A map `m` is said to contain a mapping for a key `k` if and only if `m.containsKey(k)` would return `true`.)

Parameters:

`key` - key with which the specified value is to be associated

`value` - value to be associated with the specified key

Returns:

the previous value associated with key, or null if there was no mapping for key. (A null return can also indicate that the map previously associated null with key, if the implementation supports null values.)

Throws:

`UnsupportedOperationException` - if the put operation is not supported by this map

`ClassCastException` - if the class of the specified key or value prevents it from being stored in this map

`NullPointerException` - if the specified key or value is null and this map does not permit null keys or values

`IllegalArgumentException` - if some property of the specified key or value prevents it from being stored in this map

remove

```
V remove(Object key)
```

Removes the mapping for a key from this map if it is present (optional operation). More formally, if this map contains a mapping from key `k` to value `v` such that `(key==null ? k==null : key.equals(k))`, that mapping is removed. (The map can contain at most one such mapping.)

Returns the value to which this map previously associated the key, or `null` if the map contained no mapping for the key.

If this map permits null values, then a return value of `null` does not *necessarily* indicate that the map contained no mapping for the key; it's also possible that the map explicitly mapped the key to `null`.

The map will not contain a mapping for the specified key once the call returns.

Parameters:

`key` - key whose mapping is to be removed from the map

Returns:

the previous value associated with key, or null if there was no mapping for key.

Throws:

`UnsupportedOperationException` - if the remove operation is not supported by this map

`ClassCastException` - if the key is of an inappropriate type for this map (optional)

`NullPointerException` - if the specified key is null and this map does not permit null keys (optional)

putAll

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

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 `put(k, v)` on this map once for each mapping from key `k` to value `v` in the specified map. The behavior of this operation is undefined if the specified map is modified while the operation is in progress.

Parameters:

`m - mappings to be stored in this map`

Throws:

`UnsupportedOperationException - if the putAll operation is not supported by this map`

`ClassCastException - if the class of a key or value in the specified map prevents it from being stored in this map`

`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`

`IllegalArgumentException - if some property of a key or value in the specified map prevents it from being stored in this map`

clear

```
void clear()
```

Removes all of the mappings from this map (optional operation). The map will be empty after this call returns.

Throws:

`UnsupportedOperationException - if the clear operation is not supported by this map`

keySet

```
Set<K> keySet()
```

Returns a `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 `remove` operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:

`a set view of the keys contained in this map`

values

```
Collection<V> values()
```

Returns a `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 `remove` operation), the results of the iteration are undefined. The collection supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`,

`Collection.remove`, `removeAll`, `retainAll` and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:

`a collection view of the values contained in this map`

entrySet

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

Returns a `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 `remove` operation, or through the `setValue` 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 `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll` and `clear` operations. It does not support the `add` or `addAll` operations.

Returns:

`a set view of the mappings contained in this map`

equals

```
boolean equals(Object o)
```

Compares the specified object with this map for equality. Returns `true` if the given object is also a map and the two maps represent the same mappings. More formally, two maps `m1` and `m2` represent the same mappings if `m1.entrySet().equals(m2.entrySet())`. This ensures that the `equals` method works properly across different implementations of the `Map` interface.

Overrides:

`equals` in class `Object`

Parameters:

`o - object to be compared for equality with this map`

Returns:

`true if the specified object is equal to this map`

See Also:

`Object.hashCode()`, `HashMap`

hashCode

```
int hashCode()
```

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 `entrySet()` view. This ensures that `m1.equals(m2)` implies that `m1.hashCode()==m2.hashCode()` for any two maps `m1` and `m2`, as required by the general contract of `Object.hashCode()`.

Overrides:

`hashCode` in class `Object`

Returns:

`the hash code value for this map`

See Also:

Map.Entry.hashCode(), Object.equals(Object), equals(Object)

getOrDefault

```
default V getOrDefault(Object key,
                       V defaultValue)
```

Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.

Implementation Requirements:

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.

Parameters:

key - the key whose associated value is to be returned

defaultValue - the default mapping of the key

Returns:

the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key

Throws:

ClassCastException - if the key is of an inappropriate type for this map (optional)

NullPointerException - if the specified key is null and this map does not permit null keys (optional)

Since:

1.8

forEach

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

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.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
 for (Map.Entry<K, V> entry : map.entrySet())
     action.accept(entry.getKey(), entry.getValue());
```

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.

Parameters:

action - The action to be performed for each entry

Throws:

NullPointerException - if the specified action is null

ConcurrentModificationException - if an entry is found to be removed during iteration

Since:

1.8

replaceAll

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

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.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
 for (Map.Entry<K, V> entry : map.entrySet())
     entry.setValue(function.apply(entry.getKey(), entry.getValue()));
```

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.

Parameters:

function - the function to apply to each entry

Throws:

UnsupportedOperationException - if the set operation is not supported by this map's entry set iterator.

ClassCastException - if the class of a replacement value prevents it from being stored in this map

NullPointerException - if the specified function is null, or the specified replacement value is null, and this map does not permit null values

ClassCastException - if a replacement value is of an inappropriate type for this map (optional)

NullPointerException - if function or a replacement value is null, and this map does not permit null keys or values (optional)

IllegalArgumentException - if some property of a replacement value prevents it from being stored in this map (optional)

ConcurrentModificationException - if an entry is found to be removed during iteration

Since:

1.8

putIfAbsent

```
default V putIfAbsent(K key,
                      V value)
```

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
V v = map.get(key);
if (v == null)
    v = map.put(key, value);

return v;
```

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.

Parameters:

key - key with which the specified value is to be associated

value - value to be associated with the specified key

Returns:

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the key or value is of an inappropriate type for this map (optional)

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values (optional)

IllegalArgumentException - if some property of the specified key or value prevents it from being stored in this map (optional)

Since:

1.8

remove

```
default boolean remove(Object key,
                       Object value)
```

Removes the entry for the specified key only if it is currently mapped to the specified value.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
if (map.containsKey(key) && Objects.equals(map.get(key), value)) {
    map.remove(key);
    return true;
} else
    return false;
```

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.

Parameters:

key - key with which the specified value is associated

value - value expected to be associated with the specified key

Returns:

true if the value was removed

Throws:

UnsupportedOperationException - if the remove operation is not supported by this map (optional)

ClassCastException - if the key or value is of an inappropriate type for this map (optional)

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values (optional)

Since:

1.8

replace

```
default boolean replace(K key,
                        V oldValue,
                        V newValue)
```

Replaces the entry for the specified key only if currently mapped to the specified value.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
 if (map.containsKey(key) && Objects.equals(map.get(key), value)) {
     map.put(key, newValue);
     return true;
 } else
     return false;
```

The default implementation does not throw NullPointerException for maps that do not support null values if oldValue is null unless newValue is also null.

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.

Parameters:

key - key with which the specified value is associated

oldValue - value expected to be associated with the specified key

newValue - value to be associated with the specified key

Returns:

true if the value was replaced

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of a specified key or value prevents it from being stored in this map

NullPointerException - if a specified key or newValue is null, and this map does not permit null keys or values

NullPointerException - if oldValue is null and this map does not permit null values (optional)

IllegalArgumentException - if some property of a specified key or value prevents it from being stored in this map

Since:

1.8

replace

```
default V replace(K key,
                  V value)
```

Replaces the entry for the specified key only if it is currently mapped to some value.

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
 if (map.containsKey(key)) {
     return map.put(key, value);
 } else
     return null;
```

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.

Parameters:

key - key with which the specified value is associated

value - value to be associated with the specified key

Returns:

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values

IllegalArgumentException - if some property of the specified key or value prevents it from being stored in this map

Since:

1.8

computeIfAbsent

```
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
```

If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.

If the function returns `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:

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

Or to implement a multi-value map, `Map<K,Collection<V>>`, supporting multiple values per key:

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

Implementation Requirements:

The default implementation is equivalent to the following steps for this map, then returning the current value or null if now absent:

```
if (map.get(key) == null) {
    V newValue = mappingFunction.apply(key);
    if (newValue != null)
        map.put(key, newValue);
}
```

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 `ConcurrentMap` must document whether the function is applied once atomically only if the value is not present.

Parameters:

`key` - key with which the specified value is to be associated

`mappingFunction` - the function to compute a value

Returns:

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

`UnsupportedOperationException` - if the put operation is not supported by this map (`optional`)

`ClassCastException` - if the class of the specified key or value prevents it from being stored in this map (`optional`)

Since:

1.8

computeIfPresent

```
default V computeIfPresent(K key,
                           BiFunction<? super K,? super V,? extends V> remappingFunction)
```

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.

If the function returns null, the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if now absent:

```
 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);
 }
```

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 ConcurrentMap must document whether the function is applied once atomically only if the value is not present.

Parameters:

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

Returns:

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

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

Since:

1.8

compute

```
default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remappingFunction)
```

Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). For example, to either create or append a String msg to a value mapping:

```
 map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))
```

(Method merge() is often simpler to use for such purposes.)

If the function returns 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.

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if absent:

```
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;
}
```

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 ConcurrentMap must document whether the function is applied once atomically only if the value is not present.

Parameters:

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

Returns:

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

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

Since:

1.8

merge

```
default V merge(K key,
                V value,
                BiFunction<? super V,? super V,? extends V> remappingFunction)
```

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 `null`. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a `String msg` to a value mapping:

```
map.merge(key, msg, String::concat)
```

If the function returns `null` the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if absent:

```
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);
```

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 `ConcurrentMap` must document whether the function is applied once atomically only if the value is not present.

Parameters:

`key` - key with which the resulting value is to be associated

`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

`remappingFunction` - the function to recompute a value if present

Returns:

the new value associated with the specified key, or null if no value is associated with the key

Throws:

`UnsupportedOperationException` - if the put operation is not supported by this map (`optional`)

`ClassCastException` - if the class of the specified key or value prevents it from being stored in this map (`optional`)

`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