/\*\*

\* Immutable data encourages pure functions (data-in, data-out) and lends itself

\* to much simpler application development and enabling techniques from

\* functional programming such as lazy evaluation.

\*

\* While designed to bring these powerful functional concepts to JavaScript, it

\* presents an Object-Oriented API familiar to Javascript engineers and closely

\* mirroring that of Array, Map, and Set. It is easy and efficient to convert to

\* and from plain Javascript types.

\*

\* ## How to read these docs

\*

\* In order to better explain what kinds of values the Immutable.js API expects

\* and produces, this documentation is presented in a statically typed dialect of

\* JavaScript (like [Flow][] or [TypeScript][]). You \*don't need\* to use these

\* type checking tools in order to use Immutable.js, however becoming familiar

\* with their syntax will help you get a deeper understanding of this API.

\*

\* \*\*A few examples and how to read them.\*\*

\*

\* All methods describe the kinds of data they accept and the kinds of data

\* they return. For example a function which accepts two numbers and returns

\* a number would look like this:

\*

\* ```js

\* sum(first: number, second: number): number

\* ```

\*

\* Sometimes, methods can accept different kinds of data or return different

\* kinds of data, and this is described with a \*type variable\*, which is

\* typically in all-caps. For example, a function which always returns the same

\* kind of data it was provided would look like this:

\*

\* ```js

\* identity<T>(value: T): T

\* ```

\*

\* Type variables are defined with classes and referred to in methods. For

\* example, a class that holds onto a value for you might look like this:

\*

\* ```js

\* class Box<T> {

\* constructor(value: T)

\* getValue(): T

\* }

\* ```

\*

\* In order to manipulate Immutable data, methods that we're used to affecting

\* a Collection instead return a new Collection of the same type. The type

\* `this` refers to the same kind of class. For example, a List which returns

\* new Lists when you `push` a value onto it might look like:

\*

\* ```js

\* class List<T> {

\* push(value: T): this

\* }

\* ```

\*

\* Many methods in Immutable.js accept values which implement the JavaScript

\* [Iterable][] protocol, and might appear like `Iterable<string>` for something

\* which represents sequence of strings. Typically in JavaScript we use plain

\* Arrays (`[]`) when an Iterable is expected, but also all of the Immutable.js

\* collections are iterable themselves!

\*

\* For example, to get a value deep within a structure of data, we might use

\* `getIn` which expects an `Iterable` path:

\*

\* ```

\* getIn(path: Iterable<string | number>): unknown

\* ```

\*

\* To use this method, we could pass an array: `data.getIn([ "key", 2 ])`.

\*

\*

\* Note: All examples are presented in the modern [ES2015][] version of

\* JavaScript. Use tools like Babel to support older browsers.

\*

\* For example:

\*

\* ```js

\* // ES2015

\* const mappedFoo = foo.map(x => x \* x);

\* // ES5

\* var mappedFoo = foo.map(function (x) { return x \* x; });

\* ```

\*

\* [ES2015]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New\_in\_JavaScript/ECMAScript\_6\_support\_in\_Mozilla

\* [TypeScript]: https://www.typescriptlang.org/

\* [Flow]: https://flowtype.org/

\* [Iterable]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols

\*/

declare namespace Immutable {

/\*\*

\* Lists are ordered indexed dense collections, much like a JavaScript

\* Array.

\*

\* Lists are immutable and fully persistent with O(log32 N) gets and sets,

\* and O(1) push and pop.

\*

\* Lists implement Deque, with efficient addition and removal from both the

\* end (`push`, `pop`) and beginning (`unshift`, `shift`).

\*

\* Unlike a JavaScript Array, there is no distinction between an

\* "unset" index and an index set to `undefined`. `List#forEach` visits all

\* indices from 0 to size, regardless of whether they were explicitly defined.

\*/

namespace List {

/\*\*

\* True if the provided value is a List

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable');

\* List.isList([]); // false

\* List.isList(List()); // true

\* ```

\*/

function isList(maybeList: unknown): maybeList is List<unknown>;

/\*\*

\* Creates a new List containing `values`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable');

\* List.of(1, 2, 3, 4)

\* // List [ 1, 2, 3, 4 ]

\* ```

\*

\* Note: Values are not altered or converted in any way.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable');

\* List.of({x:1}, 2, [3], 4)

\* // List [ { x: 1 }, 2, [ 3 ], 4 ]

\* ```

\*/

function of<T>(...values: Array<T>): List<T>;

}

/\*\*

\* Create a new immutable List containing the values of the provided

\* collection-like.

\*

\* Note: `List` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List, Set } = require('immutable')

\*

\* const emptyList = List()

\* // List []

\*

\* const plainArray = [ 1, 2, 3, 4 ]

\* const listFromPlainArray = List(plainArray)

\* // List [ 1, 2, 3, 4 ]

\*

\* const plainSet = Set([ 1, 2, 3, 4 ])

\* const listFromPlainSet = List(plainSet)

\* // List [ 1, 2, 3, 4 ]

\*

\* const arrayIterator = plainArray[Symbol.iterator]()

\* const listFromCollectionArray = List(arrayIterator)

\* // List [ 1, 2, 3, 4 ]

\*

\* listFromPlainArray.equals(listFromCollectionArray) // true

\* listFromPlainSet.equals(listFromCollectionArray) // true

\* listFromPlainSet.equals(listFromPlainArray) // true

\* ```

\*/

function List<T>(collection?: Iterable<T> | ArrayLike<T>): List<T>;

interface List<T> extends Collection.Indexed<T> {

/\*\*

\* The number of items in this List.

\*/

readonly size: number;

// Persistent changes

/\*\*

\* Returns a new List which includes `value` at `index`. If `index` already

\* exists in this List, it will be replaced.

\*

\* `index` may be a negative number, which indexes back from the end of the

\* List. `v.set(-1, "value")` sets the last item in the List.

\*

\* If `index` larger than `size`, the returned List's `size` will be large

\* enough to include the `index`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* const originalList = List([ 0 ]);

\* // List [ 0 ]

\* originalList.set(1, 1);

\* // List [ 0, 1 ]

\* originalList.set(0, 'overwritten');

\* // List [ "overwritten" ]

\* originalList.set(2, 2);

\* // List [ 0, undefined, 2 ]

\*

\* List().set(50000, 'value').size;

\* // 50001

\* ```

\*

\* Note: `set` can be used in `withMutations`.

\*/

set(index: number, value: T): List<T>;

/\*\*

\* Returns a new List which excludes this `index` and with a size 1 less

\* than this List. Values at indices above `index` are shifted down by 1 to

\* fill the position.

\*

\* This is synonymous with `list.splice(index, 1)`.

\*

\* `index` may be a negative number, which indexes back from the end of the

\* List. `v.delete(-1)` deletes the last item in the List.

\*

\* Note: `delete` cannot be safely used in IE8

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 0, 1, 2, 3, 4 ]).delete(0);

\* // List [ 1, 2, 3, 4 ]

\* ```

\*

\* Since `delete()` re-indexes values, it produces a complete copy, which

\* has `O(N)` complexity.

\*

\* Note: `delete` \*cannot\* be used in `withMutations`.

\*

\* @alias remove

\*/

delete(index: number): List<T>;

remove(index: number): List<T>;

/\*\*

\* Returns a new List with `value` at `index` with a size 1 more than this

\* List. Values at indices above `index` are shifted over by 1.

\*

\* This is synonymous with `list.splice(index, 0, value)`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 0, 1, 2, 3, 4 ]).insert(6, 5)

\* // List [ 0, 1, 2, 3, 4, 5 ]

\* ```

\*

\* Since `insert()` re-indexes values, it produces a complete copy, which

\* has `O(N)` complexity.

\*

\* Note: `insert` \*cannot\* be used in `withMutations`.

\*/

insert(index: number, value: T): List<T>;

/\*\*

\* Returns a new List with 0 size and no values in constant time.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 1, 2, 3, 4 ]).clear()

\* // List []

\* ```

\*

\* Note: `clear` can be used in `withMutations`.

\*/

clear(): List<T>;

/\*\*

\* Returns a new List with the provided `values` appended, starting at this

\* List's `size`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 1, 2, 3, 4 ]).push(5)

\* // List [ 1, 2, 3, 4, 5 ]

\* ```

\*

\* Note: `push` can be used in `withMutations`.

\*/

push(...values: Array<T>): List<T>;

/\*\*

\* Returns a new List with a size ones less than this List, excluding

\* the last index in this List.

\*

\* Note: this differs from `Array#pop` because it returns a new

\* List rather than the removed value. Use `last()` to get the last value

\* in this List.

\*

\* ```js

\* List([ 1, 2, 3, 4 ]).pop()

\* // List[ 1, 2, 3 ]

\* ```

\*

\* Note: `pop` can be used in `withMutations`.

\*/

pop(): List<T>;

/\*\*

\* Returns a new List with the provided `values` prepended, shifting other

\* values ahead to higher indices.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 2, 3, 4]).unshift(1);

\* // List [ 1, 2, 3, 4 ]

\* ```

\*

\* Note: `unshift` can be used in `withMutations`.

\*/

unshift(...values: Array<T>): List<T>;

/\*\*

\* Returns a new List with a size ones less than this List, excluding

\* the first index in this List, shifting all other values to a lower index.

\*

\* Note: this differs from `Array#shift` because it returns a new

\* List rather than the removed value. Use `first()` to get the first

\* value in this List.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 0, 1, 2, 3, 4 ]).shift();

\* // List [ 1, 2, 3, 4 ]

\* ```

\*

\* Note: `shift` can be used in `withMutations`.

\*/

shift(): List<T>;

/\*\*

\* Returns a new List with an updated value at `index` with the return

\* value of calling `updater` with the existing value, or `notSetValue` if

\* `index` was not set. If called with a single argument, `updater` is

\* called with the List itself.

\*

\* `index` may be a negative number, which indexes back from the end of the

\* List. `v.update(-1)` updates the last item in the List.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* const list = List([ 'a', 'b', 'c' ])

\* const result = list.update(2, val => val.toUpperCase())

\* // List [ "a", "b", "C" ]

\* ```

\*

\* This can be very useful as a way to "chain" a normal function into a

\* sequence of methods. RxJS calls this "let" and lodash calls it "thru".

\*

\* For example, to sum a List after mapping and filtering:

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* function sum(collection) {

\* return collection.reduce((sum, x) => sum + x, 0)

\* }

\*

\* List([ 1, 2, 3 ])

\* .map(x => x + 1)

\* .filter(x => x % 2 === 0)

\* .update(sum)

\* // 6

\* ```

\*

\* Note: `update(index)` can be used in `withMutations`.

\*

\* @see `Map#update`

\*/

update(index: number, notSetValue: T, updater: (value: T) => T): this;

update(index: number, updater: (value: T | undefined) => T): this;

update<R>(updater: (value: this) => R): R;

/\*\*

\* Returns a new List with size `size`. If `size` is less than this

\* List's size, the new List will exclude values at the higher indices.

\* If `size` is greater than this List's size, the new List will have

\* undefined values for the newly available indices.

\*

\* When building a new List and the final size is known up front, `setSize`

\* used in conjunction with `withMutations` may result in the more

\* performant construction.

\*/

setSize(size: number): List<T>;

// Deep persistent changes

/\*\*

\* Returns a new List having set `value` at this `keyPath`. If any keys in

\* `keyPath` do not exist, a new immutable Map will be created at that key.

\*

\* Index numbers are used as keys to determine the path to follow in

\* the List.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* const list = List([ 0, 1, 2, List([ 3, 4 ])])

\* list.setIn([3, 0], 999);

\* // List [ 0, 1, 2, List [ 999, 4 ] ]

\* ```

\*

\* Plain JavaScript Object or Arrays may be nested within an Immutable.js

\* Collection, and setIn() can update those values as well, treating them

\* immutably by creating new copies of those values with the changes applied.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* const list = List([ 0, 1, 2, { plain: 'object' }])

\* list.setIn([3, 'plain'], 'value');

\* // List([ 0, 1, 2, { plain: 'value' }])

\* ```

\*

\* Note: `setIn` can be used in `withMutations`.

\*/

setIn(keyPath: Iterable<unknown>, value: unknown): this;

/\*\*

\* Returns a new List having removed the value at this `keyPath`. If any

\* keys in `keyPath` do not exist, no change will occur.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* const list = List([ 0, 1, 2, List([ 3, 4 ])])

\* list.deleteIn([3, 0]);

\* // List [ 0, 1, 2, List [ 4 ] ]

\* ```

\*

\* Plain JavaScript Object or Arrays may be nested within an Immutable.js

\* Collection, and removeIn() can update those values as well, treating them

\* immutably by creating new copies of those values with the changes applied.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* const list = List([ 0, 1, 2, { plain: 'object' }])

\* list.removeIn([3, 'plain']);

\* // List([ 0, 1, 2, {}])

\* ```

\*

\* Note: `deleteIn` \*cannot\* be safely used in `withMutations`.

\*

\* @alias removeIn

\*/

deleteIn(keyPath: Iterable<unknown>): this;

removeIn(keyPath: Iterable<unknown>): this;

/\*\*

\* Note: `updateIn` can be used in `withMutations`.

\*

\* @see `Map#updateIn`

\*/

updateIn(

keyPath: Iterable<unknown>,

notSetValue: unknown,

updater: (value: unknown) => unknown

): this;

updateIn(

keyPath: Iterable<unknown>,

updater: (value: unknown) => unknown

): this;

/\*\*

\* Note: `mergeIn` can be used in `withMutations`.

\*

\* @see `Map#mergeIn`

\*/

mergeIn(keyPath: Iterable<unknown>, ...collections: Array<unknown>): this;

/\*\*

\* Note: `mergeDeepIn` can be used in `withMutations`.

\*

\* @see `Map#mergeDeepIn`

\*/

mergeDeepIn(

keyPath: Iterable<unknown>,

...collections: Array<unknown>

): this;

// Transient changes

/\*\*

\* Note: Not all methods can be safely used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* allows being used in `withMutations`.

\*

\* @see `Map#withMutations`

\*/

withMutations(mutator: (mutable: this) => unknown): this;

/\*\*

\* An alternative API for withMutations()

\*

\* Note: Not all methods can be safely used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* allows being used in `withMutations`.

\*

\* @see `Map#asMutable`

\*/

asMutable(): this;

/\*\*

\* @see `Map#wasAltered`

\*/

wasAltered(): boolean;

/\*\*

\* @see `Map#asImmutable`

\*/

asImmutable(): this;

// Sequence algorithms

/\*\*

\* Returns a new List with other values or collections concatenated to this one.

\*

\* Note: `concat` can be used in `withMutations`.

\*

\* @alias merge

\*/

concat<C>(...valuesOrCollections: Array<Iterable<C> | C>): List<T | C>;

merge<C>(...collections: Array<Iterable<C>>): List<T | C>;

/\*\*

\* Returns a new List with values passed through a

\* `mapper` function.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* List([ 1, 2 ]).map(x => 10 \* x)

\* // List [ 10, 20 ]

\* ```

\*/

map<M>(

mapper: (value: T, key: number, iter: this) => M,

context?: unknown

): List<M>;

/\*\*

\* Flat-maps the List, returning a new List.

\*

\* Similar to `list.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: number, iter: this) => Iterable<M>,

context?: unknown

): List<M>;

/\*\*

\* Returns a new List with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, index: number, iter: this) => value is F,

context?: unknown

): List<F>;

filter(

predicate: (value: T, index: number, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* Returns a List "zipped" with the provided collection.

\*

\* Like `zipWith`, but using the default `zipper`: creating an `Array`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 4, 5, 6 ]);

\* const c = a.zip(b); // List [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

\* ```

\*/

zip<U>(other: Collection<unknown, U>): List<[T, U]>;

zip<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): List<[T, U, V]>;

zip(...collections: Array<Collection<unknown, unknown>>): List<unknown>;

/\*\*

\* Returns a List "zipped" with the provided collections.

\*

\* Unlike `zip`, `zipAll` continues zipping until the longest collection is

\* exhausted. Missing values from shorter collections are filled with `undefined`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* const a = List([ 1, 2 ]);

\* const b = List([ 3, 4, 5 ]);

\* const c = a.zipAll(b); // List [ [ 1, 3 ], [ 2, 4 ], [ undefined, 5 ] ]

\* ```

\*

\* Note: Since zipAll will return a collection as large as the largest

\* input, some results may contain undefined values. TypeScript cannot

\* account for these without cases (as of v2.5).

\*/

zipAll<U>(other: Collection<unknown, U>): List<[T, U]>;

zipAll<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): List<[T, U, V]>;

zipAll(...collections: Array<Collection<unknown, unknown>>): List<unknown>;

/\*\*

\* Returns a List "zipped" with the provided collections by using a

\* custom `zipper` function.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable');" }

\* -->

\* ```js

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 4, 5, 6 ]);

\* const c = a.zipWith((a, b) => a + b, b);

\* // List [ 5, 7, 9 ]

\* ```

\*/

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<unknown, U>

): List<Z>;

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<unknown, U>,

thirdCollection: Collection<unknown, V>

): List<Z>;

zipWith<Z>(

zipper: (...values: Array<unknown>) => Z,

...collections: Array<Collection<unknown, unknown>>

): List<Z>;

}

/\*\*

\* Immutable Map is an unordered Collection.Keyed of (key, value) pairs with

\* `O(log32 N)` gets and `O(log32 N)` persistent sets.

\*

\* Iteration order of a Map is undefined, however is stable. Multiple

\* iterations of the same Map will iterate in the same order.

\*

\* Map's keys can be of any type, and use `Immutable.is` to determine key

\* equality. This allows the use of any value (including NaN) as a key.

\*

\* Because `Immutable.is` returns equality based on value semantics, and

\* Immutable collections are treated as values, any Immutable collection may

\* be used as a key.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, List } = require('immutable');

\* Map().set(List([ 1 ]), 'listofone').get(List([ 1 ]));

\* // 'listofone'

\* ```

\*

\* Any JavaScript object may be used as a key, however strict identity is used

\* to evaluate key equality. Two similar looking objects will represent two

\* different keys.

\*

\* Implemented by a hash-array mapped trie.

\*/

namespace Map {

/\*\*

\* True if the provided value is a Map

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map.isMap({}) // false

\* Map.isMap(Map()) // true

\* ```

\*/

function isMap(maybeMap: unknown): maybeMap is Map<unknown, unknown>;

/\*\*

\* Creates a new Map from alternating keys and values

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map.of(

\* 'key', 'value',

\* 'numerical value', 3,

\* 0, 'numerical key'

\* )

\* // Map { 0: "numerical key", "key": "value", "numerical value": 3 }

\* ```

\*

\* @deprecated Use Map([ [ 'k', 'v' ] ]) or Map({ k: 'v' })

\*/

function of(...keyValues: Array<unknown>): Map<unknown, unknown>;

}

/\*\*

\* Creates a new Immutable Map.

\*

\* Created with the same key value pairs as the provided Collection.Keyed or

\* JavaScript Object or expects a Collection of [K, V] tuple entries.

\*

\* Note: `Map` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ key: "value" })

\* Map([ [ "key", "value" ] ])

\* ```

\*

\* Keep in mind, when using JS objects to construct Immutable Maps, that

\* JavaScript Object properties are always strings, even if written in a

\* quote-less shorthand, while Immutable Maps accept keys of any type.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* let obj = { 1: "one" }

\* Object.keys(obj) // [ "1" ]

\* assert.equal(obj["1"], obj[1]) // "one" === "one"

\*

\* let map = Map(obj)

\* assert.notEqual(map.get("1"), map.get(1)) // "one" !== undefined

\* ```

\*

\* Property access for JavaScript Objects first converts the key to a string,

\* but since Immutable Map keys can be of any type the argument to `get()` is

\* not altered.

\*/

function Map<K, V>(collection?: Iterable<[K, V]>): Map<K, V>;

function Map<V>(obj: { [key: string]: V }): Map<string, V>;

function Map<K extends string | symbol, V>(obj: { [P in K]?: V }): Map<K, V>;

interface Map<K, V> extends Collection.Keyed<K, V> {

/\*\*

\* The number of entries in this Map.

\*/

readonly size: number;

// Persistent changes

/\*\*

\* Returns a new Map also containing the new key, value pair. If an equivalent

\* key already exists in this Map, it will be replaced.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const originalMap = Map()

\* const newerMap = originalMap.set('key', 'value')

\* const newestMap = newerMap.set('key', 'newer value')

\*

\* originalMap

\* // Map {}

\* newerMap

\* // Map { "key": "value" }

\* newestMap

\* // Map { "key": "newer value" }

\* ```

\*

\* Note: `set` can be used in `withMutations`.

\*/

set(key: K, value: V): this;

/\*\*

\* Returns a new Map which excludes this `key`.

\*

\* Note: `delete` cannot be safely used in IE8, but is provided to mirror

\* the ES6 collection API.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const originalMap = Map({

\* key: 'value',

\* otherKey: 'other value'

\* })

\* // Map { "key": "value", "otherKey": "other value" }

\* originalMap.delete('otherKey')

\* // Map { "key": "value" }

\* ```

\*

\* Note: `delete` can be used in `withMutations`.

\*

\* @alias remove

\*/

delete(key: K): this;

remove(key: K): this;

/\*\*

\* Returns a new Map which excludes the provided `keys`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const names = Map({ a: "Aaron", b: "Barry", c: "Connor" })

\* names.deleteAll([ 'a', 'c' ])

\* // Map { "b": "Barry" }

\* ```

\*

\* Note: `deleteAll` can be used in `withMutations`.

\*

\* @alias removeAll

\*/

deleteAll(keys: Iterable<K>): this;

removeAll(keys: Iterable<K>): this;

/\*\*

\* Returns a new Map containing no keys or values.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ key: 'value' }).clear()

\* // Map {}

\* ```

\*

\* Note: `clear` can be used in `withMutations`.

\*/

clear(): this;

/\*\*

\* Returns a new Map having updated the value at this `key` with the return

\* value of calling `updater` with the existing value.

\*

\* Similar to: `map.set(key, updater(map.get(key)))`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const aMap = Map({ key: 'value' })

\* const newMap = aMap.update('key', value => value + value)

\* // Map { "key": "valuevalue" }

\* ```

\*

\* This is most commonly used to call methods on collections within a

\* structure of data. For example, in order to `.push()` onto a nested `List`,

\* `update` and `push` can be used together:

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map, List } = require('immutable');" }

\* -->

\* ```js

\* const aMap = Map({ nestedList: List([ 1, 2, 3 ]) })

\* const newMap = aMap.update('nestedList', list => list.push(4))

\* // Map { "nestedList": List [ 1, 2, 3, 4 ] }

\* ```

\*

\* When a `notSetValue` is provided, it is provided to the `updater`

\* function when the value at the key does not exist in the Map.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* const aMap = Map({ key: 'value' })

\* const newMap = aMap.update('noKey', 'no value', value => value + value)

\* // Map { "key": "value", "noKey": "no valueno value" }

\* ```

\*

\* However, if the `updater` function returns the same value it was called

\* with, then no change will occur. This is still true if `notSetValue`

\* is provided.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* const aMap = Map({ apples: 10 })

\* const newMap = aMap.update('oranges', 0, val => val)

\* // Map { "apples": 10 }

\* assert.strictEqual(newMap, map);

\* ```

\*

\* For code using ES2015 or later, using `notSetValue` is discourged in

\* favor of function parameter default values. This helps to avoid any

\* potential confusion with identify functions as described above.

\*

\* The previous example behaves differently when written with default values:

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* const aMap = Map({ apples: 10 })

\* const newMap = aMap.update('oranges', (val = 0) => val)

\* // Map { "apples": 10, "oranges": 0 }

\* ```

\*

\* If no key is provided, then the `updater` function return value is

\* returned as well.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* const aMap = Map({ key: 'value' })

\* const result = aMap.update(aMap => aMap.get('key'))

\* // "value"

\* ```

\*

\* This can be very useful as a way to "chain" a normal function into a

\* sequence of methods. RxJS calls this "let" and lodash calls it "thru".

\*

\* For example, to sum the values in a Map

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable');" }

\* -->

\* ```js

\* function sum(collection) {

\* return collection.reduce((sum, x) => sum + x, 0)

\* }

\*

\* Map({ x: 1, y: 2, z: 3 })

\* .map(x => x + 1)

\* .filter(x => x % 2 === 0)

\* .update(sum)

\* // 6

\* ```

\*

\* Note: `update(key)` can be used in `withMutations`.

\*/

update(key: K, notSetValue: V, updater: (value: V) => V): this;

update(key: K, updater: (value: V | undefined) => V): this;

update<R>(updater: (value: this) => R): R;

/\*\*

\* Returns a new Map resulting from merging the provided Collections

\* (or JS objects) into this Map. In other words, this takes each entry of

\* each collection and sets it on this Map.

\*

\* Note: Values provided to `merge` are shallowly converted before being

\* merged. No nested values are altered.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const one = Map({ a: 10, b: 20, c: 30 })

\* const two = Map({ b: 40, a: 50, d: 60 })

\* one.merge(two) // Map { "a": 50, "b": 40, "c": 30, "d": 60 }

\* two.merge(one) // Map { "b": 20, "a": 10, "d": 60, "c": 30 }

\* ```

\*

\* Note: `merge` can be used in `withMutations`.

\*

\* @alias concat

\*/

merge<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): Map<K | KC, V | VC>;

merge<C>(

...collections: Array<{ [key: string]: C }>

): Map<K | string, V | C>;

concat<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): Map<K | KC, V | VC>;

concat<C>(

...collections: Array<{ [key: string]: C }>

): Map<K | string, V | C>;

/\*\*

\* Like `merge()`, `mergeWith()` returns a new Map resulting from merging

\* the provided Collections (or JS objects) into this Map, but uses the

\* `merger` function for dealing with conflicts.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const one = Map({ a: 10, b: 20, c: 30 })

\* const two = Map({ b: 40, a: 50, d: 60 })

\* one.mergeWith((oldVal, newVal) => oldVal / newVal, two)

\* // { "a": 0.2, "b": 0.5, "c": 30, "d": 60 }

\* two.mergeWith((oldVal, newVal) => oldVal / newVal, one)

\* // { "b": 2, "a": 5, "d": 60, "c": 30 }

\* ```

\*

\* Note: `mergeWith` can be used in `withMutations`.

\*/

mergeWith(

merger: (oldVal: V, newVal: V, key: K) => V,

...collections: Array<Iterable<[K, V]> | { [key: string]: V }>

): this;

/\*\*

\* Like `merge()`, but when two compatible collections are encountered with

\* the same key, it merges them as well, recursing deeply through the nested

\* data. Two collections are considered to be compatible (and thus will be

\* merged together) if they both fall into one of three categories: keyed

\* (e.g., `Map`s, `Record`s, and objects), indexed (e.g., `List`s and

\* arrays), or set-like (e.g., `Set`s). If they fall into separate

\* categories, `mergeDeep` will replace the existing collection with the

\* collection being merged in. This behavior can be customized by using

\* `mergeDeepWith()`.

\*

\* Note: Indexed and set-like collections are merged using

\* `concat()`/`union()` and therefore do not recurse.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const one = Map({ a: Map({ x: 10, y: 10 }), b: Map({ x: 20, y: 50 }) })

\* const two = Map({ a: Map({ x: 2 }), b: Map({ y: 5 }), c: Map({ z: 3 }) })

\* one.mergeDeep(two)

\* // Map {

\* // "a": Map { "x": 2, "y": 10 },

\* // "b": Map { "x": 20, "y": 5 },

\* // "c": Map { "z": 3 }

\* // }

\* ```

\*

\* Note: `mergeDeep` can be used in `withMutations`.

\*/

mergeDeep(

...collections: Array<Iterable<[K, V]> | { [key: string]: V }>

): this;

/\*\*

\* Like `mergeDeep()`, but when two non-collections or incompatible

\* collections are encountered at the same key, it uses the `merger`

\* function to determine the resulting value. Collections are considered

\* incompatible if they fall into separate categories between keyed,

\* indexed, and set-like.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const one = Map({ a: Map({ x: 10, y: 10 }), b: Map({ x: 20, y: 50 }) })

\* const two = Map({ a: Map({ x: 2 }), b: Map({ y: 5 }), c: Map({ z: 3 }) })

\* one.mergeDeepWith((oldVal, newVal) => oldVal / newVal, two)

\* // Map {

\* // "a": Map { "x": 5, "y": 10 },

\* // "b": Map { "x": 20, "y": 10 },

\* // "c": Map { "z": 3 }

\* // }

\* ```

\*

\* Note: `mergeDeepWith` can be used in `withMutations`.

\*/

mergeDeepWith(

merger: (oldVal: unknown, newVal: unknown, key: unknown) => unknown,

...collections: Array<Iterable<[K, V]> | { [key: string]: V }>

): this;

// Deep persistent changes

/\*\*

\* Returns a new Map having set `value` at this `keyPath`. If any keys in

\* `keyPath` do not exist, a new immutable Map will be created at that key.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const originalMap = Map({

\* subObject: Map({

\* subKey: 'subvalue',

\* subSubObject: Map({

\* subSubKey: 'subSubValue'

\* })

\* })

\* })

\*

\* const newMap = originalMap.setIn(['subObject', 'subKey'], 'ha ha!')

\* // Map {

\* // "subObject": Map {

\* // "subKey": "ha ha!",

\* // "subSubObject": Map { "subSubKey": "subSubValue" }

\* // }

\* // }

\*

\* const newerMap = originalMap.setIn(

\* ['subObject', 'subSubObject', 'subSubKey'],

\* 'ha ha ha!'

\* )

\* // Map {

\* // "subObject": Map {

\* // "subKey": "subvalue",

\* // "subSubObject": Map { "subSubKey": "ha ha ha!" }

\* // }

\* // }

\* ```

\*

\* Plain JavaScript Object or Arrays may be nested within an Immutable.js

\* Collection, and setIn() can update those values as well, treating them

\* immutably by creating new copies of those values with the changes applied.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const originalMap = Map({

\* subObject: {

\* subKey: 'subvalue',

\* subSubObject: {

\* subSubKey: 'subSubValue'

\* }

\* }

\* })

\*

\* originalMap.setIn(['subObject', 'subKey'], 'ha ha!')

\* // Map {

\* // "subObject": {

\* // subKey: "ha ha!",

\* // subSubObject: { subSubKey: "subSubValue" }

\* // }

\* // }

\* ```

\*

\* If any key in the path exists but cannot be updated (such as a primitive

\* like number or a custom Object like Date), an error will be thrown.

\*

\* Note: `setIn` can be used in `withMutations`.

\*/

setIn(keyPath: Iterable<unknown>, value: unknown): this;

/\*\*

\* Returns a new Map having removed the value at this `keyPath`. If any keys

\* in `keyPath` do not exist, no change will occur.

\*

\* Note: `deleteIn` can be used in `withMutations`.

\*

\* @alias removeIn

\*/

deleteIn(keyPath: Iterable<unknown>): this;

removeIn(keyPath: Iterable<unknown>): this;

/\*\*

\* Returns a new Map having applied the `updater` to the entry found at the

\* keyPath.

\*

\* This is most commonly used to call methods on collections nested within a

\* structure of data. For example, in order to `.push()` onto a nested `List`,

\* `updateIn` and `push` can be used together:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, List } = require('immutable')

\* const map = Map({ inMap: Map({ inList: List([ 1, 2, 3 ]) }) })

\* const newMap = map.updateIn(['inMap', 'inList'], list => list.push(4))

\* // Map { "inMap": Map { "inList": List [ 1, 2, 3, 4 ] } }

\* ```

\*

\* If any keys in `keyPath` do not exist, new Immutable `Map`s will

\* be created at those keys. If the `keyPath` does not already contain a

\* value, the `updater` function will be called with `notSetValue`, if

\* provided, otherwise `undefined`.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable')" }

\* -->

\* ```js

\* const map = Map({ a: Map({ b: Map({ c: 10 }) }) })

\* const newMap = map.updateIn(['a', 'b', 'c'], val => val \* 2)

\* // Map { "a": Map { "b": Map { "c": 20 } } }

\* ```

\*

\* If the `updater` function returns the same value it was called with, then

\* no change will occur. This is still true if `notSetValue` is provided.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable')" }

\* -->

\* ```js

\* const map = Map({ a: Map({ b: Map({ c: 10 }) }) })

\* const newMap = map.updateIn(['a', 'b', 'x'], 100, val => val)

\* // Map { "a": Map { "b": Map { "c": 10 } } }

\* assert.strictEqual(newMap, aMap)

\* ```

\*

\* For code using ES2015 or later, using `notSetValue` is discourged in

\* favor of function parameter default values. This helps to avoid any

\* potential confusion with identify functions as described above.

\*

\* The previous example behaves differently when written with default values:

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable')" }

\* -->

\* ```js

\* const map = Map({ a: Map({ b: Map({ c: 10 }) }) })

\* const newMap = map.updateIn(['a', 'b', 'x'], (val = 100) => val)

\* // Map { "a": Map { "b": Map { "c": 10, "x": 100 } } }

\* ```

\*

\* Plain JavaScript Object or Arrays may be nested within an Immutable.js

\* Collection, and updateIn() can update those values as well, treating them

\* immutably by creating new copies of those values with the changes applied.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Map } = require('immutable')" }

\* -->

\* ```js

\* const map = Map({ a: { b: { c: 10 } } })

\* const newMap = map.updateIn(['a', 'b', 'c'], val => val \* 2)

\* // Map { "a": { b: { c: 20 } } }

\* ```

\*

\* If any key in the path exists but cannot be updated (such as a primitive

\* like number or a custom Object like Date), an error will be thrown.

\*

\* Note: `updateIn` can be used in `withMutations`.

\*/

updateIn(

keyPath: Iterable<unknown>,

notSetValue: unknown,

updater: (value: unknown) => unknown

): this;

updateIn(

keyPath: Iterable<unknown>,

updater: (value: unknown) => unknown

): this;

/\*\*

\* A combination of `updateIn` and `merge`, returning a new Map, but

\* performing the merge at a point arrived at by following the keyPath.

\* In other words, these two lines are equivalent:

\*

\* ```js

\* map.updateIn(['a', 'b', 'c'], abc => abc.merge(y))

\* map.mergeIn(['a', 'b', 'c'], y)

\* ```

\*

\* Note: `mergeIn` can be used in `withMutations`.

\*/

mergeIn(keyPath: Iterable<unknown>, ...collections: Array<unknown>): this;

/\*\*

\* A combination of `updateIn` and `mergeDeep`, returning a new Map, but

\* performing the deep merge at a point arrived at by following the keyPath.

\* In other words, these two lines are equivalent:

\*

\* ```js

\* map.updateIn(['a', 'b', 'c'], abc => abc.mergeDeep(y))

\* map.mergeDeepIn(['a', 'b', 'c'], y)

\* ```

\*

\* Note: `mergeDeepIn` can be used in `withMutations`.

\*/

mergeDeepIn(

keyPath: Iterable<unknown>,

...collections: Array<unknown>

): this;

// Transient changes

/\*\*

\* Every time you call one of the above functions, a new immutable Map is

\* created. If a pure function calls a number of these to produce a final

\* return value, then a penalty on performance and memory has been paid by

\* creating all of the intermediate immutable Maps.

\*

\* If you need to apply a series of mutations to produce a new immutable

\* Map, `withMutations()` creates a temporary mutable copy of the Map which

\* can apply mutations in a highly performant manner. In fact, this is

\* exactly how complex mutations like `merge` are done.

\*

\* As an example, this results in the creation of 2, not 4, new Maps:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const map1 = Map()

\* const map2 = map1.withMutations(map => {

\* map.set('a', 1).set('b', 2).set('c', 3)

\* })

\* assert.equal(map1.size, 0)

\* assert.equal(map2.size, 3)

\* ```

\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Read the documentation for each method to see if it

\* is safe to use in `withMutations`.

\*/

withMutations(mutator: (mutable: this) => unknown): this;

/\*\*

\* Another way to avoid creation of intermediate Immutable maps is to create

\* a mutable copy of this collection. Mutable copies \*always\* return `this`,

\* and thus shouldn't be used for equality. Your function should never return

\* a mutable copy of a collection, only use it internally to create a new

\* collection.

\*

\* If possible, use `withMutations` to work with temporary mutable copies as

\* it provides an easier to use API and considers many common optimizations.

\*

\* Note: if the collection is already mutable, `asMutable` returns itself.

\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Read the documentation for each method to see if it

\* is safe to use in `withMutations`.

\*

\* @see `Map#asImmutable`

\*/

asMutable(): this;

/\*\*

\* Returns true if this is a mutable copy (see `asMutable()`) and mutative

\* alterations have been applied.

\*

\* @see `Map#asMutable`

\*/

wasAltered(): boolean;

/\*\*

\* The yin to `asMutable`'s yang. Because it applies to mutable collections,

\* this operation is \*mutable\* and may return itself (though may not

\* return itself, i.e. if the result is an empty collection). Once

\* performed, the original mutable copy must no longer be mutated since it

\* may be the immutable result.

\*

\* If possible, use `withMutations` to work with temporary mutable copies as

\* it provides an easier to use API and considers many common optimizations.

\*

\* @see `Map#asMutable`

\*/

asImmutable(): this;

// Sequence algorithms

/\*\*

\* Returns a new Map with values passed through a

\* `mapper` function.

\*

\* Map({ a: 1, b: 2 }).map(x => 10 \* x)

\* // Map { a: 10, b: 20 }

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Map<K, M>;

/\*\*

\* @see Collection.Keyed.mapKeys

\*/

mapKeys<M>(

mapper: (key: K, value: V, iter: this) => M,

context?: unknown

): Map<M, V>;

/\*\*

\* @see Collection.Keyed.mapEntries

\*/

mapEntries<KM, VM>(

mapper: (

entry: [K, V],

index: number,

iter: this

) => [KM, VM] | undefined,

context?: unknown

): Map<KM, VM>;

/\*\*

\* Flat-maps the Map, returning a new Map.

\*

\* Similar to `data.map(...).flatten(true)`.

\*/

flatMap<KM, VM>(

mapper: (value: V, key: K, iter: this) => Iterable<[KM, VM]>,

context?: unknown

): Map<KM, VM>;

/\*\*

\* Returns a new Map with only the entries for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): Map<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* @see Collection.Keyed.flip

\*/

flip(): Map<V, K>;

}

/\*\*

\* A type of Map that has the additional guarantee that the iteration order of

\* entries will be the order in which they were set().

\*

\* The iteration behavior of OrderedMap is the same as native ES6 Map and

\* JavaScript Object.

\*

\* Note that `OrderedMap` are more expensive than non-ordered `Map` and may

\* consume more memory. `OrderedMap#set` is amortized O(log32 N), but not

\* stable.

\*/

namespace OrderedMap {

/\*\*

\* True if the provided value is an OrderedMap.

\*/

function isOrderedMap(

maybeOrderedMap: unknown

): maybeOrderedMap is OrderedMap<unknown, unknown>;

}

/\*\*

\* Creates a new Immutable OrderedMap.

\*

\* Created with the same key value pairs as the provided Collection.Keyed or

\* JavaScript Object or expects a Collection of [K, V] tuple entries.

\*

\* The iteration order of key-value pairs provided to this constructor will

\* be preserved in the OrderedMap.

\*

\* let newOrderedMap = OrderedMap({key: "value"})

\* let newOrderedMap = OrderedMap([["key", "value"]])

\*

\* Note: `OrderedMap` is a factory function and not a class, and does not use

\* the `new` keyword during construction.

\*/

function OrderedMap<K, V>(collection?: Iterable<[K, V]>): OrderedMap<K, V>;

function OrderedMap<V>(obj: { [key: string]: V }): OrderedMap<string, V>;

interface OrderedMap<K, V> extends Map<K, V> {

/\*\*

\* The number of entries in this OrderedMap.

\*/

readonly size: number;

/\*\*

\* Returns a new OrderedMap also containing the new key, value pair. If an

\* equivalent key already exists in this OrderedMap, it will be replaced

\* while maintaining the existing order.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { OrderedMap } = require('immutable')

\* const originalMap = OrderedMap({a:1, b:1, c:1})

\* const updatedMap = originalMap.set('b', 2)

\*

\* originalMap

\* // OrderedMap {a: 1, b: 1, c: 1}

\* updatedMap

\* // OrderedMap {a: 1, b: 2, c: 1}

\* ```

\*

\* Note: `set` can be used in `withMutations`.

\*/

set(key: K, value: V): this;

/\*\*

\* Returns a new OrderedMap resulting from merging the provided Collections

\* (or JS objects) into this OrderedMap. In other words, this takes each

\* entry of each collection and sets it on this OrderedMap.

\*

\* Note: Values provided to `merge` are shallowly converted before being

\* merged. No nested values are altered.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { OrderedMap } = require('immutable')

\* const one = OrderedMap({ a: 10, b: 20, c: 30 })

\* const two = OrderedMap({ b: 40, a: 50, d: 60 })

\* one.merge(two) // OrderedMap { "a": 50, "b": 40, "c": 30, "d": 60 }

\* two.merge(one) // OrderedMap { "b": 20, "a": 10, "d": 60, "c": 30 }

\* ```

\*

\* Note: `merge` can be used in `withMutations`.

\*

\* @alias concat

\*/

merge<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): OrderedMap<K | KC, V | VC>;

merge<C>(

...collections: Array<{ [key: string]: C }>

): OrderedMap<K | string, V | C>;

concat<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): OrderedMap<K | KC, V | VC>;

concat<C>(

...collections: Array<{ [key: string]: C }>

): OrderedMap<K | string, V | C>;

// Sequence algorithms

/\*\*

\* Returns a new OrderedMap with values passed through a

\* `mapper` function.

\*

\* OrderedMap({ a: 1, b: 2 }).map(x => 10 \* x)

\* // OrderedMap { "a": 10, "b": 20 }

\*

\* Note: `map()` always returns a new instance, even if it produced the same

\* value at every step.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): OrderedMap<K, M>;

/\*\*

\* @see Collection.Keyed.mapKeys

\*/

mapKeys<M>(

mapper: (key: K, value: V, iter: this) => M,

context?: unknown

): OrderedMap<M, V>;

/\*\*

\* @see Collection.Keyed.mapEntries

\*/

mapEntries<KM, VM>(

mapper: (

entry: [K, V],

index: number,

iter: this

) => [KM, VM] | undefined,

context?: unknown

): OrderedMap<KM, VM>;

/\*\*

\* Flat-maps the OrderedMap, returning a new OrderedMap.

\*

\* Similar to `data.map(...).flatten(true)`.

\*/

flatMap<KM, VM>(

mapper: (value: V, key: K, iter: this) => Iterable<[KM, VM]>,

context?: unknown

): OrderedMap<KM, VM>;

/\*\*

\* Returns a new OrderedMap with only the entries for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): OrderedMap<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* @see Collection.Keyed.flip

\*/

flip(): OrderedMap<V, K>;

}

/\*\*

\* A Collection of unique values with `O(log32 N)` adds and has.

\*

\* When iterating a Set, the entries will be (value, value) pairs. Iteration

\* order of a Set is undefined, however is stable. Multiple iterations of the

\* same Set will iterate in the same order.

\*

\* Set values, like Map keys, may be of any type. Equality is determined using

\* `Immutable.is`, enabling Sets to uniquely include other Immutable

\* collections, custom value types, and NaN.

\*/

namespace Set {

/\*\*

\* True if the provided value is a Set

\*/

function isSet(maybeSet: unknown): maybeSet is Set<unknown>;

/\*\*

\* Creates a new Set containing `values`.

\*/

function of<T>(...values: Array<T>): Set<T>;

/\*\*

\* `Set.fromKeys()` creates a new immutable Set containing the keys from

\* this Collection or JavaScript Object.

\*/

function fromKeys<T>(iter: Collection<T, unknown>): Set<T>;

function fromKeys(obj: { [key: string]: unknown }): Set<string>;

/\*\*

\* `Set.intersect()` creates a new immutable Set that is the intersection of

\* a collection of other sets.

\*

\* ```js

\* const { Set } = require('immutable')

\* const intersected = Set.intersect([

\* Set([ 'a', 'b', 'c' ])

\* Set([ 'c', 'a', 't' ])

\* ])

\* // Set [ "a", "c" ]

\* ```

\*/

function intersect<T>(sets: Iterable<Iterable<T>>): Set<T>;

/\*\*

\* `Set.union()` creates a new immutable Set that is the union of a

\* collection of other sets.

\*

\* ```js

\* const { Set } = require('immutable')

\* const unioned = Set.union([

\* Set([ 'a', 'b', 'c' ])

\* Set([ 'c', 'a', 't' ])

\* ])

\* // Set [ "a", "b", "c", "t" ]

\* ```

\*/

function union<T>(sets: Iterable<Iterable<T>>): Set<T>;

}

/\*\*

\* Create a new immutable Set containing the values of the provided

\* collection-like.

\*

\* Note: `Set` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*/

function Set<T>(collection?: Iterable<T> | ArrayLike<T>): Set<T>;

interface Set<T> extends Collection.Set<T> {

/\*\*

\* The number of items in this Set.

\*/

readonly size: number;

// Persistent changes

/\*\*

\* Returns a new Set which also includes this value.

\*

\* Note: `add` can be used in `withMutations`.

\*/

add(value: T): this;

/\*\*

\* Returns a new Set which excludes this value.

\*

\* Note: `delete` can be used in `withMutations`.

\*

\* Note: `delete` \*\*cannot\*\* be safely used in IE8, use `remove` if

\* supporting old browsers.

\*

\* @alias remove

\*/

delete(value: T): this;

remove(value: T): this;

/\*\*

\* Returns a new Set containing no values.

\*

\* Note: `clear` can be used in `withMutations`.

\*/

clear(): this;

/\*\*

\* Returns a Set including any value from `collections` that does not already

\* exist in this Set.

\*

\* Note: `union` can be used in `withMutations`.

\* @alias merge

\* @alias concat

\*/

union<C>(...collections: Array<Iterable<C>>): Set<T | C>;

merge<C>(...collections: Array<Iterable<C>>): Set<T | C>;

concat<C>(...collections: Array<Iterable<C>>): Set<T | C>;

/\*\*

\* Returns a Set which has removed any values not also contained

\* within `collections`.

\*

\* Note: `intersect` can be used in `withMutations`.

\*/

intersect(...collections: Array<Iterable<T>>): this;

/\*\*

\* Returns a Set excluding any values contained within `collections`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { OrderedSet } = require('immutable')

\* OrderedSet([ 1, 2, 3 ]).subtract([1, 3])

\* // OrderedSet [2]

\* ```

\*

\* Note: `subtract` can be used in `withMutations`.

\*/

subtract(...collections: Array<Iterable<T>>): this;

// Transient changes

/\*\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* mentions being safe to use in `withMutations`.

\*

\* @see `Map#withMutations`

\*/

withMutations(mutator: (mutable: this) => unknown): this;

/\*\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* mentions being safe to use in `withMutations`.

\*

\* @see `Map#asMutable`

\*/

asMutable(): this;

/\*\*

\* @see `Map#wasAltered`

\*/

wasAltered(): boolean;

/\*\*

\* @see `Map#asImmutable`

\*/

asImmutable(): this;

// Sequence algorithms

/\*\*

\* Returns a new Set with values passed through a

\* `mapper` function.

\*

\* Set([1,2]).map(x => 10 \* x)

\* // Set [10,20]

\*/

map<M>(

mapper: (value: T, key: T, iter: this) => M,

context?: unknown

): Set<M>;

/\*\*

\* Flat-maps the Set, returning a new Set.

\*

\* Similar to `set.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: T, iter: this) => Iterable<M>,

context?: unknown

): Set<M>;

/\*\*

\* Returns a new Set with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, key: T, iter: this) => value is F,

context?: unknown

): Set<F>;

filter(

predicate: (value: T, key: T, iter: this) => unknown,

context?: unknown

): this;

}

/\*\*

\* A type of Set that has the additional guarantee that the iteration order of

\* values will be the order in which they were `add`ed.

\*

\* The iteration behavior of OrderedSet is the same as native ES6 Set.

\*

\* Note that `OrderedSet` are more expensive than non-ordered `Set` and may

\* consume more memory. `OrderedSet#add` is amortized O(log32 N), but not

\* stable.

\*/

namespace OrderedSet {

/\*\*

\* True if the provided value is an OrderedSet.

\*/

function isOrderedSet(maybeOrderedSet: unknown): boolean;

/\*\*

\* Creates a new OrderedSet containing `values`.

\*/

function of<T>(...values: Array<T>): OrderedSet<T>;

/\*\*

\* `OrderedSet.fromKeys()` creates a new immutable OrderedSet containing

\* the keys from this Collection or JavaScript Object.

\*/

function fromKeys<T>(iter: Collection<T, unknown>): OrderedSet<T>;

function fromKeys(obj: { [key: string]: unknown }): OrderedSet<string>;

}

/\*\*

\* Create a new immutable OrderedSet containing the values of the provided

\* collection-like.

\*

\* Note: `OrderedSet` is a factory function and not a class, and does not use

\* the `new` keyword during construction.

\*/

function OrderedSet<T>(

collection?: Iterable<T> | ArrayLike<T>

): OrderedSet<T>;

interface OrderedSet<T> extends Set<T> {

/\*\*

\* The number of items in this OrderedSet.

\*/

readonly size: number;

/\*\*

\* Returns an OrderedSet including any value from `collections` that does

\* not already exist in this OrderedSet.

\*

\* Note: `union` can be used in `withMutations`.

\* @alias merge

\* @alias concat

\*/

union<C>(...collections: Array<Iterable<C>>): OrderedSet<T | C>;

merge<C>(...collections: Array<Iterable<C>>): OrderedSet<T | C>;

concat<C>(...collections: Array<Iterable<C>>): OrderedSet<T | C>;

// Sequence algorithms

/\*\*

\* Returns a new Set with values passed through a

\* `mapper` function.

\*

\* OrderedSet([ 1, 2 ]).map(x => 10 \* x)

\* // OrderedSet [10, 20]

\*/

map<M>(

mapper: (value: T, key: T, iter: this) => M,

context?: unknown

): OrderedSet<M>;

/\*\*

\* Flat-maps the OrderedSet, returning a new OrderedSet.

\*

\* Similar to `set.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: T, iter: this) => Iterable<M>,

context?: unknown

): OrderedSet<M>;

/\*\*

\* Returns a new OrderedSet with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, key: T, iter: this) => value is F,

context?: unknown

): OrderedSet<F>;

filter(

predicate: (value: T, key: T, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* Returns an OrderedSet of the same type "zipped" with the provided

\* collections.

\*

\* Like `zipWith`, but using the default `zipper`: creating an `Array`.

\*

\* ```js

\* const a = OrderedSet([ 1, 2, 3 ])

\* const b = OrderedSet([ 4, 5, 6 ])

\* const c = a.zip(b)

\* // OrderedSet [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

\* ```

\*/

zip<U>(other: Collection<unknown, U>): OrderedSet<[T, U]>;

zip<U, V>(

other1: Collection<unknown, U>,

other2: Collection<unknown, V>

): OrderedSet<[T, U, V]>;

zip(

...collections: Array<Collection<unknown, unknown>>

): OrderedSet<unknown>;

/\*\*

\* Returns a OrderedSet of the same type "zipped" with the provided

\* collections.

\*

\* Unlike `zip`, `zipAll` continues zipping until the longest collection is

\* exhausted. Missing values from shorter collections are filled with `undefined`.

\*

\* ```js

\* const a = OrderedSet([ 1, 2 ]);

\* const b = OrderedSet([ 3, 4, 5 ]);

\* const c = a.zipAll(b); // OrderedSet [ [ 1, 3 ], [ 2, 4 ], [ undefined, 5 ] ]

\* ```

\*

\* Note: Since zipAll will return a collection as large as the largest

\* input, some results may contain undefined values. TypeScript cannot

\* account for these without cases (as of v2.5).

\*/

zipAll<U>(other: Collection<unknown, U>): OrderedSet<[T, U]>;

zipAll<U, V>(

other1: Collection<unknown, U>,

other2: Collection<unknown, V>

): OrderedSet<[T, U, V]>;

zipAll(

...collections: Array<Collection<unknown, unknown>>

): OrderedSet<unknown>;

/\*\*

\* Returns an OrderedSet of the same type "zipped" with the provided

\* collections by using a custom `zipper` function.

\*

\* @see Seq.Indexed.zipWith

\*/

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<unknown, U>

): OrderedSet<Z>;

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<unknown, U>,

thirdCollection: Collection<unknown, V>

): OrderedSet<Z>;

zipWith<Z>(

zipper: (...values: Array<unknown>) => Z,

...collections: Array<Collection<unknown, unknown>>

): OrderedSet<Z>;

}

/\*\*

\* Stacks are indexed collections which support very efficient O(1) addition

\* and removal from the front using `unshift(v)` and `shift()`.

\*

\* For familiarity, Stack also provides `push(v)`, `pop()`, and `peek()`, but

\* be aware that they also operate on the front of the list, unlike List or

\* a JavaScript Array.

\*

\* Note: `reverse()` or any inherent reverse traversal (`reduceRight`,

\* `lastIndexOf`, etc.) is not efficient with a Stack.

\*

\* Stack is implemented with a Single-Linked List.

\*/

namespace Stack {

/\*\*

\* True if the provided value is a Stack

\*/

function isStack(maybeStack: unknown): maybeStack is Stack<unknown>;

/\*\*

\* Creates a new Stack containing `values`.

\*/

function of<T>(...values: Array<T>): Stack<T>;

}

/\*\*

\* Create a new immutable Stack containing the values of the provided

\* collection-like.

\*

\* The iteration order of the provided collection is preserved in the

\* resulting `Stack`.

\*

\* Note: `Stack` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*/

function Stack<T>(collection?: Iterable<T> | ArrayLike<T>): Stack<T>;

interface Stack<T> extends Collection.Indexed<T> {

/\*\*

\* The number of items in this Stack.

\*/

readonly size: number;

// Reading values

/\*\*

\* Alias for `Stack.first()`.

\*/

peek(): T | undefined;

// Persistent changes

/\*\*

\* Returns a new Stack with 0 size and no values.

\*

\* Note: `clear` can be used in `withMutations`.

\*/

clear(): Stack<T>;

/\*\*

\* Returns a new Stack with the provided `values` prepended, shifting other

\* values ahead to higher indices.

\*

\* This is very efficient for Stack.

\*

\* Note: `unshift` can be used in `withMutations`.

\*/

unshift(...values: Array<T>): Stack<T>;

/\*\*

\* Like `Stack#unshift`, but accepts a collection rather than varargs.

\*

\* Note: `unshiftAll` can be used in `withMutations`.

\*/

unshiftAll(iter: Iterable<T>): Stack<T>;

/\*\*

\* Returns a new Stack with a size ones less than this Stack, excluding

\* the first item in this Stack, shifting all other values to a lower index.

\*

\* Note: this differs from `Array#shift` because it returns a new

\* Stack rather than the removed value. Use `first()` or `peek()` to get the

\* first value in this Stack.

\*

\* Note: `shift` can be used in `withMutations`.

\*/

shift(): Stack<T>;

/\*\*

\* Alias for `Stack#unshift` and is not equivalent to `List#push`.

\*/

push(...values: Array<T>): Stack<T>;

/\*\*

\* Alias for `Stack#unshiftAll`.

\*/

pushAll(iter: Iterable<T>): Stack<T>;

/\*\*

\* Alias for `Stack#shift` and is not equivalent to `List#pop`.

\*/

pop(): Stack<T>;

// Transient changes

/\*\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* mentions being safe to use in `withMutations`.

\*

\* @see `Map#withMutations`

\*/

withMutations(mutator: (mutable: this) => unknown): this;

/\*\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Check the documentation for each method to see if it

\* mentions being safe to use in `withMutations`.

\*

\* @see `Map#asMutable`

\*/

asMutable(): this;

/\*\*

\* @see `Map#wasAltered`

\*/

wasAltered(): boolean;

/\*\*

\* @see `Map#asImmutable`

\*/

asImmutable(): this;

// Sequence algorithms

/\*\*

\* Returns a new Stack with other collections concatenated to this one.

\*/

concat<C>(...valuesOrCollections: Array<Iterable<C> | C>): Stack<T | C>;

/\*\*

\* Returns a new Stack with values passed through a

\* `mapper` function.

\*

\* Stack([ 1, 2 ]).map(x => 10 \* x)

\* // Stack [ 10, 20 ]

\*

\* Note: `map()` always returns a new instance, even if it produced the same

\* value at every step.

\*/

map<M>(

mapper: (value: T, key: number, iter: this) => M,

context?: unknown

): Stack<M>;

/\*\*

\* Flat-maps the Stack, returning a new Stack.

\*

\* Similar to `stack.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: number, iter: this) => Iterable<M>,

context?: unknown

): Stack<M>;

/\*\*

\* Returns a new Set with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, index: number, iter: this) => value is F,

context?: unknown

): Set<F>;

filter(

predicate: (value: T, index: number, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* Returns a Stack "zipped" with the provided collections.

\*

\* Like `zipWith`, but using the default `zipper`: creating an `Array`.

\*

\* ```js

\* const a = Stack([ 1, 2, 3 ]);

\* const b = Stack([ 4, 5, 6 ]);

\* const c = a.zip(b); // Stack [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

\* ```

\*/

zip<U>(other: Collection<unknown, U>): Stack<[T, U]>;

zip<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Stack<[T, U, V]>;

zip(...collections: Array<Collection<unknown, unknown>>): Stack<unknown>;

/\*\*

\* Returns a Stack "zipped" with the provided collections.

\*

\* Unlike `zip`, `zipAll` continues zipping until the longest collection is

\* exhausted. Missing values from shorter collections are filled with `undefined`.

\*

\* ```js

\* const a = Stack([ 1, 2 ]);

\* const b = Stack([ 3, 4, 5 ]);

\* const c = a.zipAll(b); // Stack [ [ 1, 3 ], [ 2, 4 ], [ undefined, 5 ] ]

\* ```

\*

\* Note: Since zipAll will return a collection as large as the largest

\* input, some results may contain undefined values. TypeScript cannot

\* account for these without cases (as of v2.5).

\*/

zipAll<U>(other: Collection<unknown, U>): Stack<[T, U]>;

zipAll<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Stack<[T, U, V]>;

zipAll(...collections: Array<Collection<unknown, unknown>>): Stack<unknown>;

/\*\*

\* Returns a Stack "zipped" with the provided collections by using a

\* custom `zipper` function.

\*

\* ```js

\* const a = Stack([ 1, 2, 3 ]);

\* const b = Stack([ 4, 5, 6 ]);

\* const c = a.zipWith((a, b) => a + b, b);

\* // Stack [ 5, 7, 9 ]

\* ```

\*/

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<unknown, U>

): Stack<Z>;

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<unknown, U>,

thirdCollection: Collection<unknown, V>

): Stack<Z>;

zipWith<Z>(

zipper: (...values: Array<unknown>) => Z,

...collections: Array<Collection<unknown, unknown>>

): Stack<Z>;

}

/\*\*

\* Returns a Seq.Indexed of numbers from `start` (inclusive) to `end`

\* (exclusive), by `step`, where `start` defaults to 0, `step` to 1, and `end` to

\* infinity. When `start` is equal to `end`, returns empty range.

\*

\* Note: `Range` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*

\* ```js

\* const { Range } = require('immutable')

\* Range() // [ 0, 1, 2, 3, ... ]

\* Range(10) // [ 10, 11, 12, 13, ... ]

\* Range(10, 15) // [ 10, 11, 12, 13, 14 ]

\* Range(10, 30, 5) // [ 10, 15, 20, 25 ]

\* Range(30, 10, 5) // [ 30, 25, 20, 15 ]

\* Range(30, 30, 5) // []

\* ```

\*/

function Range(

start?: number,

end?: number,

step?: number

): Seq.Indexed<number>;

/\*\*

\* Returns a Seq.Indexed of `value` repeated `times` times. When `times` is

\* not defined, returns an infinite `Seq` of `value`.

\*

\* Note: `Repeat` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*

\* ```js

\* const { Repeat } = require('immutable')

\* Repeat('foo') // [ 'foo', 'foo', 'foo', ... ]

\* Repeat('bar', 4) // [ 'bar', 'bar', 'bar', 'bar' ]

\* ```

\*/

function Repeat<T>(value: T, times?: number): Seq.Indexed<T>;

/\*\*

\* A record is similar to a JS object, but enforces a specific set of allowed

\* string keys, and has default values.

\*

\* The `Record()` function produces new Record Factories, which when called

\* create Record instances.

\*

\* ```js

\* const { Record } = require('immutable')

\* const ABRecord = Record({ a: 1, b: 2 })

\* const myRecord = ABRecord({ b: 3 })

\* ```

\*

\* Records always have a value for the keys they define. `remove`ing a key

\* from a record simply resets it to the default value for that key.

\*

\* ```js

\* myRecord.get('a') // 1

\* myRecord.get('b') // 3

\* const myRecordWithoutB = myRecord.remove('b')

\* myRecordWithoutB.get('b') // 2

\* ```

\*

\* Values provided to the constructor not found in the Record type will

\* be ignored. For example, in this case, ABRecord is provided a key "x" even

\* though only "a" and "b" have been defined. The value for "x" will be

\* ignored for this record.

\*

\* ```js

\* const myRecord = ABRecord({ b: 3, x: 10 })

\* myRecord.get('x') // undefined

\* ```

\*

\* Because Records have a known set of string keys, property get access works

\* as expected, however property sets will throw an Error.

\*

\* Note: IE8 does not support property access. Only use `get()` when

\* supporting IE8.

\*

\* ```js

\* myRecord.b // 3

\* myRecord.b = 5 // throws Error

\* ```

\*

\* Record Types can be extended as well, allowing for custom methods on your

\* Record. This is not a common pattern in functional environments, but is in

\* many JS programs.

\*

\* However Record Types are more restricted than typical JavaScript classes.

\* They do not use a class constructor, which also means they cannot use

\* class properties (since those are technically part of a constructor).

\*

\* While Record Types can be syntactically created with the JavaScript `class`

\* form, the resulting Record function is actually a factory function, not a

\* class constructor. Even though Record Types are not classes, JavaScript

\* currently requires the use of `new` when creating new Record instances if

\* they are defined as a `class`.

\*

\* ```

\* class ABRecord extends Record({ a: 1, b: 2 }) {

\* getAB() {

\* return this.a + this.b;

\* }

\* }

\*

\* var myRecord = new ABRecord({b: 3})

\* myRecord.getAB() // 4

\* ```

\*

\*

\* \*\*Flow Typing Records:\*\*

\*

\* Immutable.js exports two Flow types designed to make it easier to use

\* Records with flow typed code, `RecordOf<TProps>` and `RecordFactory<TProps>`.

\*

\* When defining a new kind of Record factory function, use a flow type that

\* describes the values the record contains along with `RecordFactory<TProps>`.

\* To type instances of the Record (which the factory function returns),

\* use `RecordOf<TProps>`.

\*

\* Typically, new Record definitions will export both the Record factory

\* function as well as the Record instance type for use in other code.

\*

\* ```js

\* import type { RecordFactory, RecordOf } from 'immutable';

\*

\* // Use RecordFactory<TProps> for defining new Record factory functions.

\* type Point3DProps = { x: number, y: number, z: number };

\* const defaultValues: Point3DProps = { x: 0, y: 0, z: 0 };

\* const makePoint3D: RecordFactory<Point3DProps> = Record(defaultValues);

\* export makePoint3D;

\*

\* // Use RecordOf<T> for defining new instances of that Record.

\* export type Point3D = RecordOf<Point3DProps>;

\* const some3DPoint: Point3D = makePoint3D({ x: 10, y: 20, z: 30 });

\* ```

\*

\* \*\*Flow Typing Record Subclasses:\*\*

\*

\* Records can be subclassed as a means to add additional methods to Record

\* instances. This is generally discouraged in favor of a more functional API,

\* since Subclasses have some minor overhead. However the ability to create

\* a rich API on Record types can be quite valuable.

\*

\* When using Flow to type Subclasses, do not use `RecordFactory<TProps>`,

\* instead apply the props type when subclassing:

\*

\* ```js

\* type PersonProps = {name: string, age: number};

\* const defaultValues: PersonProps = {name: 'Aristotle', age: 2400};

\* const PersonRecord = Record(defaultValues);

\* class Person extends PersonRecord<PersonProps> {

\* getName(): string {

\* return this.get('name')

\* }

\*

\* setName(name: string): this {

\* return this.set('name', name);

\* }

\* }

\* ```

\*

\* \*\*Choosing Records vs plain JavaScript objects\*\*

\*

\* Records offer a persistently immutable alternative to plain JavaScript

\* objects, however they're not required to be used within Immutable.js

\* collections. In fact, the deep-access and deep-updating functions

\* like `getIn()` and `setIn()` work with plain JavaScript Objects as well.

\*

\* Deciding to use Records or Objects in your application should be informed

\* by the tradeoffs and relative benefits of each:

\*

\* - \*Runtime immutability\*: plain JS objects may be carefully treated as

\* immutable, however Record instances will \*throw\* if attempted to be

\* mutated directly. Records provide this additional guarantee, however at

\* some marginal runtime cost. While JS objects are mutable by nature, the

\* use of type-checking tools like [Flow](https://medium.com/@gcanti/immutability-with-flow-faa050a1aef4)

\* can help gain confidence in code written to favor immutability.

\*

\* - \*Value equality\*: Records use value equality when compared with `is()`

\* or `record.equals()`. That is, two Records with the same keys and values

\* are equal. Plain objects use \*reference equality\*. Two objects with the

\* same keys and values are not equal since they are different objects.

\* This is important to consider when using objects as keys in a `Map` or

\* values in a `Set`, which use equality when retrieving values.

\*

\* - \*API methods\*: Records have a full featured API, with methods like

\* `.getIn()`, and `.equals()`. These can make working with these values

\* easier, but comes at the cost of not allowing keys with those names.

\*

\* - \*Default values\*: Records provide default values for every key, which

\* can be useful when constructing Records with often unchanging values.

\* However default values can make using Flow and TypeScript more laborious.

\*

\* - \*Serialization\*: Records use a custom internal representation to

\* efficiently store and update their values. Converting to and from this

\* form isn't free. If converting Records to plain objects is common,

\* consider sticking with plain objects to begin with.

\*/

namespace Record {

/\*\*

\* True if `maybeRecord` is an instance of a Record.

\*/

function isRecord(maybeRecord: unknown): maybeRecord is Record<{}>;

/\*\*

\* Records allow passing a second parameter to supply a descriptive name

\* that appears when converting a Record to a string or in any error

\* messages. A descriptive name for any record can be accessed by using this

\* method. If one was not provided, the string "Record" is returned.

\*

\* ```js

\* const { Record } = require('immutable')

\* const Person = Record({

\* name: null

\* }, 'Person')

\*

\* var me = Person({ name: 'My Name' })

\* me.toString() // "Person { "name": "My Name" }"

\* Record.getDescriptiveName(me) // "Person"

\* ```

\*/

function getDescriptiveName(record: Record<any>): string;

/\*\*

\* A Record.Factory is created by the `Record()` function. Record instances

\* are created by passing it some of the accepted values for that Record

\* type:

\*

\* <!-- runkit:activate

\* { "preamble": "const { Record } = require('immutable')" }

\* -->

\* ```js

\* // makePerson is a Record Factory function

\* const makePerson = Record({ name: null, favoriteColor: 'unknown' });

\*

\* // alan is a Record instance

\* const alan = makePerson({ name: 'Alan' });

\* ```

\*

\* Note that Record Factories return `Record<TProps> & Readonly<TProps>`,

\* this allows use of both the Record instance API, and direct property

\* access on the resulting instances:

\*

\* <!-- runkit:activate

\* { "preamble": "const { Record } = require('immutable');const makePerson = Record({ name: null, favoriteColor: 'unknown' });const alan = makePerson({ name: 'Alan' });" }

\* -->

\* ```js

\* // Use the Record API

\* console.log('Record API: ' + alan.get('name'))

\*

\* // Or direct property access (Readonly)

\* console.log('property access: ' + alan.name)

\* ```

\*

\* \*\*Flow Typing Records:\*\*

\*

\* Use the `RecordFactory<TProps>` Flow type to get high quality type checking of

\* Records:

\*

\* ```js

\* import type { RecordFactory, RecordOf } from 'immutable';

\*

\* // Use RecordFactory<TProps> for defining new Record factory functions.

\* type PersonProps = { name: ?string, favoriteColor: string };

\* const makePerson: RecordFactory<PersonProps> = Record({ name: null, favoriteColor: 'unknown' });

\*

\* // Use RecordOf<T> for defining new instances of that Record.

\* type Person = RecordOf<PersonProps>;

\* const alan: Person = makePerson({ name: 'Alan' });

\* ```

\*/

namespace Factory {}

interface Factory<TProps extends object> {

(values?: Partial<TProps> | Iterable<[string, unknown]>): Record<TProps> &

Readonly<TProps>;

new (

values?: Partial<TProps> | Iterable<[string, unknown]>

): Record<TProps> & Readonly<TProps>;

/\*\*

\* The name provided to `Record(values, name)` can be accessed with

\* `displayName`.

\*/

displayName: string;

}

function Factory<TProps extends object>(

values?: Partial<TProps> | Iterable<[string, unknown]>

): Record<TProps> & Readonly<TProps>;

}

/\*\*

\* Unlike other types in Immutable.js, the `Record()` function creates a new

\* Record Factory, which is a function that creates Record instances.

\*

\* See above for examples of using `Record()`.

\*

\* Note: `Record` is a factory function and not a class, and does not use the

\* `new` keyword during construction.

\*/

function Record<TProps extends object>(

defaultValues: TProps,

name?: string

): Record.Factory<TProps>;

interface Record<TProps extends object> {

// Reading values

has(key: string): key is keyof TProps & string;

/\*\*

\* Returns the value associated with the provided key, which may be the

\* default value defined when creating the Record factory function.

\*

\* If the requested key is not defined by this Record type, then

\* notSetValue will be returned if provided. Note that this scenario would

\* produce an error when using Flow or TypeScript.

\*/

get<K extends keyof TProps>(key: K, notSetValue?: unknown): TProps[K];

get<T>(key: string, notSetValue: T): T;

// Reading deep values

hasIn(keyPath: Iterable<unknown>): boolean;

getIn(keyPath: Iterable<unknown>): unknown;

// Value equality

equals(other: unknown): boolean;

hashCode(): number;

// Persistent changes

set<K extends keyof TProps>(key: K, value: TProps[K]): this;

update<K extends keyof TProps>(

key: K,

updater: (value: TProps[K]) => TProps[K]

): this;

merge(

...collections: Array<Partial<TProps> | Iterable<[string, unknown]>>

): this;

mergeDeep(

...collections: Array<Partial<TProps> | Iterable<[string, unknown]>>

): this;

mergeWith(

merger: (oldVal: unknown, newVal: unknown, key: keyof TProps) => unknown,

...collections: Array<Partial<TProps> | Iterable<[string, unknown]>>

): this;

mergeDeepWith(

merger: (oldVal: unknown, newVal: unknown, key: unknown) => unknown,

...collections: Array<Partial<TProps> | Iterable<[string, unknown]>>

): this;

/\*\*

\* Returns a new instance of this Record type with the value for the

\* specific key set to its default value.

\*

\* @alias remove

\*/

delete<K extends keyof TProps>(key: K): this;

remove<K extends keyof TProps>(key: K): this;

/\*\*

\* Returns a new instance of this Record type with all values set

\* to their default values.

\*/

clear(): this;

// Deep persistent changes

setIn(keyPath: Iterable<unknown>, value: unknown): this;

updateIn(

keyPath: Iterable<unknown>,

updater: (value: unknown) => unknown

): this;

mergeIn(keyPath: Iterable<unknown>, ...collections: Array<unknown>): this;

mergeDeepIn(

keyPath: Iterable<unknown>,

...collections: Array<unknown>

): this;

/\*\*

\* @alias removeIn

\*/

deleteIn(keyPath: Iterable<unknown>): this;

removeIn(keyPath: Iterable<unknown>): this;

// Conversion to JavaScript types

/\*\*

\* Deeply converts this Record to equivalent native JavaScript Object.

\*

\* Note: This method may not be overridden. Objects with custom

\* serialization to plain JS may override toJSON() instead.

\*/

toJS(): { [K in keyof TProps]: unknown };

/\*\*

\* Shallowly converts this Record to equivalent native JavaScript Object.

\*/

toJSON(): TProps;

/\*\*

\* Shallowly converts this Record to equivalent JavaScript Object.

\*/

toObject(): TProps;

// Transient changes

/\*\*

\* Note: Not all methods can be used on a mutable collection or within

\* `withMutations`! Only `set` may be used mutatively.

\*

\* @see `Map#withMutations`

\*/

withMutations(mutator: (mutable: this) => unknown): this;

/\*\*

\* @see `Map#asMutable`

\*/

asMutable(): this;

/\*\*

\* @see `Map#wasAltered`

\*/

wasAltered(): boolean;

/\*\*

\* @see `Map#asImmutable`

\*/

asImmutable(): this;

// Sequence algorithms

toSeq(): Seq.Keyed<keyof TProps, TProps[keyof TProps]>;

[Symbol.iterator](): IterableIterator<[keyof TProps, TProps[keyof TProps]]>;

}

/\*\*

\* RecordOf<T> is used in TypeScript to define interfaces expecting an

\* instance of record with type T.

\*

\* This is equivalent to an instance of a record created by a Record Factory.

\*/

type RecordOf<TProps extends object> = Record<TProps> & Readonly<TProps>;

/\*\*

\* `Seq` describes a lazy operation, allowing them to efficiently chain

\* use of all the higher-order collection methods (such as `map` and `filter`)

\* by not creating intermediate collections.

\*

\* \*\*Seq is immutable\*\* — Once a Seq is created, it cannot be

\* changed, appended to, rearranged or otherwise modified. Instead, any

\* mutative method called on a `Seq` will return a new `Seq`.

\*

\* \*\*Seq is lazy\*\* — `Seq` does as little work as necessary to respond to any

\* method call. Values are often created during iteration, including implicit

\* iteration when reducing or converting to a concrete data structure such as

\* a `List` or JavaScript `Array`.

\*

\* For example, the following performs no work, because the resulting

\* `Seq`'s values are never iterated:

\*

\* ```js

\* const { Seq } = require('immutable')

\* const oddSquares = Seq([ 1, 2, 3, 4, 5, 6, 7, 8 ])

\* .filter(x => x % 2 !== 0)

\* .map(x => x \* x)

\* ```

\*

\* Once the `Seq` is used, it performs only the work necessary. In this

\* example, no intermediate arrays are ever created, filter is called three

\* times, and map is only called once:

\*

\* ```js

\* oddSquares.get(1); // 9

\* ```

\*

\* Any collection can be converted to a lazy Seq with `Seq()`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const map = Map({ a: 1, b: 2, c: 3 })

\* const lazySeq = Seq(map)

\* ```

\*

\* `Seq` allows for the efficient chaining of operations, allowing for the

\* expression of logic that can otherwise be very tedious:

\*

\* ```js

\* lazySeq

\* .flip()

\* .map(key => key.toUpperCase())

\* .flip()

\* // Seq { A: 1, B: 1, C: 1 }

\* ```

\*

\* As well as expressing logic that would otherwise seem memory or time

\* limited, for example `Range` is a special kind of Lazy sequence.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Range } = require('immutable')

\* Range(1, Infinity)

\* .skip(1000)

\* .map(n => -n)

\* .filter(n => n % 2 === 0)

\* .take(2)

\* .reduce((r, n) => r \* n, 1)

\* // 1006008

\* ```

\*

\* Seq is often used to provide a rich collection API to JavaScript Object.

\*

\* ```js

\* Seq({ x: 0, y: 1, z: 2 }).map(v => v \* 2).toObject();

\* // { x: 0, y: 2, z: 4 }

\* ```

\*/

namespace Seq {

/\*\*

\* True if `maybeSeq` is a Seq, it is not backed by a concrete

\* structure such as Map, List, or Set.

\*/

function isSeq(

maybeSeq: unknown

): maybeSeq is

| Seq.Indexed<unknown>

| Seq.Keyed<unknown, unknown>

| Seq.Set<unknown>;

/\*\*

\* `Seq` which represents key-value pairs.

\*/

namespace Keyed {}

/\*\*

\* Always returns a Seq.Keyed, if input is not keyed, expects an

\* collection of [K, V] tuples.

\*

\* Note: `Seq.Keyed` is a conversion function and not a class, and does not

\* use the `new` keyword during construction.

\*/

function Keyed<K, V>(collection?: Iterable<[K, V]>): Seq.Keyed<K, V>;

function Keyed<V>(obj: { [key: string]: V }): Seq.Keyed<string, V>;

interface Keyed<K, V> extends Seq<K, V>, Collection.Keyed<K, V> {

/\*\*

\* Deeply converts this Keyed Seq to equivalent native JavaScript Object.

\*

\* Converts keys to Strings.

\*/

toJS(): { [key: string]: unknown };

/\*\*

\* Shallowly converts this Keyed Seq to equivalent native JavaScript Object.

\*

\* Converts keys to Strings.

\*/

toJSON(): { [key: string]: V };

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<[K, V]>;

/\*\*

\* Returns itself

\*/

toSeq(): this;

/\*\*

\* Returns a new Seq with other collections concatenated to this one.

\*

\* All entries will be present in the resulting Seq, even if they

\* have the same key.

\*/

concat<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): Seq.Keyed<K | KC, V | VC>;

concat<C>(

...collections: Array<{ [key: string]: C }>

): Seq.Keyed<K | string, V | C>;

/\*\*

\* Returns a new Seq.Keyed with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Seq } = require('immutable')

\* Seq.Keyed({ a: 1, b: 2 }).map(x => 10 \* x)

\* // Seq { "a": 10, "b": 20 }

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Seq.Keyed<K, M>;

/\*\*

\* @see Collection.Keyed.mapKeys

\*/

mapKeys<M>(

mapper: (key: K, value: V, iter: this) => M,

context?: unknown

): Seq.Keyed<M, V>;

/\*\*

\* @see Collection.Keyed.mapEntries

\*/

mapEntries<KM, VM>(

mapper: (

entry: [K, V],

index: number,

iter: this

) => [KM, VM] | undefined,

context?: unknown

): Seq.Keyed<KM, VM>;

/\*\*

\* Flat-maps the Seq, returning a Seq of the same type.

\*

\* Similar to `seq.map(...).flatten(true)`.

\*/

flatMap<KM, VM>(

mapper: (value: V, key: K, iter: this) => Iterable<[KM, VM]>,

context?: unknown

): Seq.Keyed<KM, VM>;

/\*\*

\* Returns a new Seq with only the entries for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): Seq.Keyed<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* @see Collection.Keyed.flip

\*/

flip(): Seq.Keyed<V, K>;

[Symbol.iterator](): IterableIterator<[K, V]>;

}

/\*\*

\* `Seq` which represents an ordered indexed list of values.

\*/

namespace Indexed {

/\*\*

\* Provides an Seq.Indexed of the values provided.

\*/

function of<T>(...values: Array<T>): Seq.Indexed<T>;

}

/\*\*

\* Always returns Seq.Indexed, discarding associated keys and

\* supplying incrementing indices.

\*

\* Note: `Seq.Indexed` is a conversion function and not a class, and does

\* not use the `new` keyword during construction.

\*/

function Indexed<T>(

collection?: Iterable<T> | ArrayLike<T>

): Seq.Indexed<T>;

interface Indexed<T> extends Seq<number, T>, Collection.Indexed<T> {

/\*\*

\* Deeply converts this Indexed Seq to equivalent native JavaScript Array.

\*/

toJS(): Array<unknown>;

/\*\*

\* Shallowly converts this Indexed Seq to equivalent native JavaScript Array.

\*/

toJSON(): Array<T>;

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<T>;

/\*\*

\* Returns itself

\*/

toSeq(): this;

/\*\*

\* Returns a new Seq with other collections concatenated to this one.

\*/

concat<C>(

...valuesOrCollections: Array<Iterable<C> | C>

): Seq.Indexed<T | C>;

/\*\*

\* Returns a new Seq.Indexed with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Seq } = require('immutable')

\* Seq.Indexed([ 1, 2 ]).map(x => 10 \* x)

\* // Seq [ 10, 20 ]

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: T, key: number, iter: this) => M,

context?: unknown

): Seq.Indexed<M>;

/\*\*

\* Flat-maps the Seq, returning a a Seq of the same type.

\*

\* Similar to `seq.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: number, iter: this) => Iterable<M>,

context?: unknown

): Seq.Indexed<M>;

/\*\*

\* Returns a new Seq with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, index: number, iter: this) => value is F,

context?: unknown

): Seq.Indexed<F>;

filter(

predicate: (value: T, index: number, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* Returns a Seq "zipped" with the provided collections.

\*

\* Like `zipWith`, but using the default `zipper`: creating an `Array`.

\*

\* ```js

\* const a = Seq([ 1, 2, 3 ]);

\* const b = Seq([ 4, 5, 6 ]);

\* const c = a.zip(b); // Seq [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

\* ```

\*/

zip<U>(other: Collection<unknown, U>): Seq.Indexed<[T, U]>;

zip<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Seq.Indexed<[T, U, V]>;

zip(

...collections: Array<Collection<unknown, unknown>>

): Seq.Indexed<unknown>;

/\*\*

\* Returns a Seq "zipped" with the provided collections.

\*

\* Unlike `zip`, `zipAll` continues zipping until the longest collection is

\* exhausted. Missing values from shorter collections are filled with `undefined`.

\*

\* ```js

\* const a = Seq([ 1, 2 ]);

\* const b = Seq([ 3, 4, 5 ]);

\* const c = a.zipAll(b); // Seq [ [ 1, 3 ], [ 2, 4 ], [ undefined, 5 ] ]

\* ```

\*/

zipAll<U>(other: Collection<unknown, U>): Seq.Indexed<[T, U]>;

zipAll<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Seq.Indexed<[T, U, V]>;

zipAll(

...collections: Array<Collection<unknown, unknown>>

): Seq.Indexed<unknown>;

/\*\*

\* Returns a Seq "zipped" with the provided collections by using a

\* custom `zipper` function.

\*

\* ```js

\* const a = Seq([ 1, 2, 3 ]);

\* const b = Seq([ 4, 5, 6 ]);

\* const c = a.zipWith((a, b) => a + b, b);

\* // Seq [ 5, 7, 9 ]

\* ```

\*/

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<unknown, U>

): Seq.Indexed<Z>;

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<unknown, U>,

thirdCollection: Collection<unknown, V>

): Seq.Indexed<Z>;

zipWith<Z>(

zipper: (...values: Array<unknown>) => Z,

...collections: Array<Collection<unknown, unknown>>

): Seq.Indexed<Z>;

[Symbol.iterator](): IterableIterator<T>;

}

/\*\*

\* `Seq` which represents a set of values.

\*

\* Because `Seq` are often lazy, `Seq.Set` does not provide the same guarantee

\* of value uniqueness as the concrete `Set`.

\*/

namespace Set {

/\*\*

\* Returns a Seq.Set of the provided values

\*/

function of<T>(...values: Array<T>): Seq.Set<T>;

}

/\*\*

\* Always returns a Seq.Set, discarding associated indices or keys.

\*

\* Note: `Seq.Set` is a conversion function and not a class, and does not

\* use the `new` keyword during construction.

\*/

function Set<T>(collection?: Iterable<T> | ArrayLike<T>): Seq.Set<T>;

interface Set<T> extends Seq<T, T>, Collection.Set<T> {

/\*\*

\* Deeply converts this Set Seq to equivalent native JavaScript Array.

\*/

toJS(): Array<unknown>;

/\*\*

\* Shallowly converts this Set Seq to equivalent native JavaScript Array.

\*/

toJSON(): Array<T>;

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<T>;

/\*\*

\* Returns itself

\*/

toSeq(): this;

/\*\*

\* Returns a new Seq with other collections concatenated to this one.

\*

\* All entries will be present in the resulting Seq, even if they

\* are duplicates.

\*/

concat<U>(...collections: Array<Iterable<U>>): Seq.Set<T | U>;

/\*\*

\* Returns a new Seq.Set with values passed through a

\* `mapper` function.

\*

\* ```js

\* Seq.Set([ 1, 2 ]).map(x => 10 \* x)

\* // Seq { 10, 20 }

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: T, key: T, iter: this) => M,

context?: unknown

): Seq.Set<M>;

/\*\*

\* Flat-maps the Seq, returning a Seq of the same type.

\*

\* Similar to `seq.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: T, iter: this) => Iterable<M>,

context?: unknown

): Seq.Set<M>;

/\*\*

\* Returns a new Seq with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, key: T, iter: this) => value is F,

context?: unknown

): Seq.Set<F>;

filter(

predicate: (value: T, key: T, iter: this) => unknown,

context?: unknown

): this;

[Symbol.iterator](): IterableIterator<T>;

}

}

/\*\*

\* Creates a Seq.

\*

\* Returns a particular kind of `Seq` based on the input.

\*

\* \* If a `Seq`, that same `Seq`.

\* \* If an `Collection`, a `Seq` of the same kind (Keyed, Indexed, or Set).

\* \* If an Array-like, an `Seq.Indexed`.

\* \* If an Iterable Object, an `Seq.Indexed`.

\* \* If an Object, a `Seq.Keyed`.

\*

\* Note: An Iterator itself will be treated as an object, becoming a `Seq.Keyed`,

\* which is usually not what you want. You should turn your Iterator Object into

\* an iterable object by defining a Symbol.iterator (or @@iterator) method which

\* returns `this`.

\*

\* Note: `Seq` is a conversion function and not a class, and does not use the

\* `new` keyword during construction.

\*/

function Seq<S extends Seq<unknown, unknown>>(seq: S): S;

function Seq<K, V>(collection: Collection.Keyed<K, V>): Seq.Keyed<K, V>;

function Seq<T>(collection: Collection.Set<T>): Seq.Set<T>;

function Seq<T>(

collection: Collection.Indexed<T> | Iterable<T> | ArrayLike<T>

): Seq.Indexed<T>;

function Seq<V>(obj: { [key: string]: V }): Seq.Keyed<string, V>;

function Seq<K = unknown, V = unknown>(): Seq<K, V>;

interface Seq<K, V> extends Collection<K, V> {

/\*\*

\* Some Seqs can describe their size lazily. When this is the case,

\* size will be an integer. Otherwise it will be undefined.

\*

\* For example, Seqs returned from `map()` or `reverse()`

\* preserve the size of the original `Seq` while `filter()` does not.

\*

\* Note: `Range`, `Repeat` and `Seq`s made from `Array`s and `Object`s will

\* always have a size.

\*/

readonly size: number | undefined;

// Force evaluation

/\*\*

\* Because Sequences are lazy and designed to be chained together, they do

\* not cache their results. For example, this map function is called a total

\* of 6 times, as each `join` iterates the Seq of three values.

\*

\* var squares = Seq([ 1, 2, 3 ]).map(x => x \* x)

\* squares.join() + squares.join()

\*

\* If you know a `Seq` will be used multiple times, it may be more

\* efficient to first cache it in memory. Here, the map function is called

\* only 3 times.

\*

\* var squares = Seq([ 1, 2, 3 ]).map(x => x \* x).cacheResult()

\* squares.join() + squares.join()

\*

\* Use this method judiciously, as it must fully evaluate a Seq which can be

\* a burden on memory and possibly performance.

\*

\* Note: after calling `cacheResult`, a Seq will always have a `size`.

\*/

cacheResult(): this;

// Sequence algorithms

/\*\*

\* Returns a new Seq with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Seq } = require('immutable')

\* Seq([ 1, 2 ]).map(x => 10 \* x)

\* // Seq [ 10, 20 ]

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the same

\* value at every step.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Seq<K, M>;

/\*\*

\* Returns a new Seq with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Seq } = require('immutable')

\* Seq([ 1, 2 ]).map(x => 10 \* x)

\* // Seq [ 10, 20 ]

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the same

\* value at every step.

\* Note: used only for sets.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Seq<M, M>;

/\*\*

\* Flat-maps the Seq, returning a Seq of the same type.

\*

\* Similar to `seq.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: V, key: K, iter: this) => Iterable<M>,

context?: unknown

): Seq<K, M>;

/\*\*

\* Flat-maps the Seq, returning a Seq of the same type.

\*

\* Similar to `seq.map(...).flatten(true)`.

\* Note: Used only for sets.

\*/

flatMap<M>(

mapper: (value: V, key: K, iter: this) => Iterable<M>,

context?: unknown

): Seq<M, M>;

/\*\*

\* Returns a new Seq with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): Seq<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

}

/\*\*

\* The `Collection` is a set of (key, value) entries which can be iterated, and

\* is the base class for all collections in `immutable`, allowing them to

\* make use of all the Collection methods (such as `map` and `filter`).

\*

\* Note: A collection is always iterated in the same order, however that order

\* may not always be well defined, as is the case for the `Map` and `Set`.

\*

\* Collection is the abstract base class for concrete data structures. It

\* cannot be constructed directly.

\*

\* Implementations should extend one of the subclasses, `Collection.Keyed`,

\* `Collection.Indexed`, or `Collection.Set`.

\*/

namespace Collection {

/\*\*

\* @deprecated use `const { isKeyed } = require('immutable')`

\*/

function isKeyed(

maybeKeyed: unknown

): maybeKeyed is Collection.Keyed<unknown, unknown>;

/\*\*

\* @deprecated use `const { isIndexed } = require('immutable')`

\*/

function isIndexed(

maybeIndexed: unknown

): maybeIndexed is Collection.Indexed<unknown>;

/\*\*

\* @deprecated use `const { isAssociative } = require('immutable')`

\*/

function isAssociative(

maybeAssociative: unknown

): maybeAssociative is

| Collection.Keyed<unknown, unknown>

| Collection.Indexed<unknown>;

/\*\*

\* @deprecated use `const { isOrdered } = require('immutable')`

\*/

function isOrdered(maybeOrdered: unknown): boolean;

/\*\*

\* Keyed Collections have discrete keys tied to each value.

\*

\* When iterating `Collection.Keyed`, each iteration will yield a `[K, V]`

\* tuple, in other words, `Collection#entries` is the default iterator for

\* Keyed Collections.

\*/

namespace Keyed {}

/\*\*

\* Creates a Collection.Keyed

\*

\* Similar to `Collection()`, however it expects collection-likes of [K, V]

\* tuples if not constructed from a Collection.Keyed or JS Object.

\*

\* Note: `Collection.Keyed` is a conversion function and not a class, and

\* does not use the `new` keyword during construction.

\*/

function Keyed<K, V>(collection?: Iterable<[K, V]>): Collection.Keyed<K, V>;

function Keyed<V>(obj: { [key: string]: V }): Collection.Keyed<string, V>;

interface Keyed<K, V> extends Collection<K, V> {

/\*\*

\* Deeply converts this Keyed collection to equivalent native JavaScript Object.

\*

\* Converts keys to Strings.

\*/

toJS(): { [key: string]: unknown };

/\*\*

\* Shallowly converts this Keyed collection to equivalent native JavaScript Object.

\*

\* Converts keys to Strings.

\*/

toJSON(): { [key: string]: V };

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<[K, V]>;

/\*\*

\* Returns Seq.Keyed.

\* @override

\*/

toSeq(): Seq.Keyed<K, V>;

// Sequence functions

/\*\*

\* Returns a new Collection.Keyed of the same type where the keys and values

\* have been flipped.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ a: 'z', b: 'y' }).flip()

\* // Map { "z": "a", "y": "b" }

\* ```

\*/

flip(): Collection.Keyed<V, K>;

/\*\*

\* Returns a new Collection with other collections concatenated to this one.

\*/

concat<KC, VC>(

...collections: Array<Iterable<[KC, VC]>>

): Collection.Keyed<K | KC, V | VC>;

concat<C>(

...collections: Array<{ [key: string]: C }>

): Collection.Keyed<K | string, V | C>;

/\*\*

\* Returns a new Collection.Keyed with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Collection } = require('immutable')

\* Collection.Keyed({ a: 1, b: 2 }).map(x => 10 \* x)

\* // Seq { "a": 10, "b": 20 }

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Collection.Keyed<K, M>;

/\*\*

\* Returns a new Collection.Keyed of the same type with keys passed through

\* a `mapper` function.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ a: 1, b: 2 }).mapKeys(x => x.toUpperCase())

\* // Map { "A": 1, "B": 2 }

\* ```

\*

\* Note: `mapKeys()` always returns a new instance, even if it produced

\* the same key at every step.

\*/

mapKeys<M>(

mapper: (key: K, value: V, iter: this) => M,

context?: unknown

): Collection.Keyed<M, V>;

/\*\*

\* Returns a new Collection.Keyed of the same type with entries

\* ([key, value] tuples) passed through a `mapper` function.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ a: 1, b: 2 })

\* .mapEntries(([ k, v ]) => [ k.toUpperCase(), v \* 2 ])

\* // Map { "A": 2, "B": 4 }

\* ```

\*

\* Note: `mapEntries()` always returns a new instance, even if it produced

\* the same entry at every step.

\*

\* If the mapper function returns `undefined`, then the entry will be filtered

\*/

mapEntries<KM, VM>(

mapper: (

entry: [K, V],

index: number,

iter: this

) => [KM, VM] | undefined,

context?: unknown

): Collection.Keyed<KM, VM>;

/\*\*

\* Flat-maps the Collection, returning a Collection of the same type.

\*

\* Similar to `collection.map(...).flatten(true)`.

\*/

flatMap<KM, VM>(

mapper: (value: V, key: K, iter: this) => Iterable<[KM, VM]>,

context?: unknown

): Collection.Keyed<KM, VM>;

/\*\*

\* Returns a new Collection with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): Collection.Keyed<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

[Symbol.iterator](): IterableIterator<[K, V]>;

}

/\*\*

\* Indexed Collections have incrementing numeric keys. They exhibit

\* slightly different behavior than `Collection.Keyed` for some methods in order

\* to better mirror the behavior of JavaScript's `Array`, and add methods

\* which do not make sense on non-indexed Collections such as `indexOf`.

\*

\* Unlike JavaScript arrays, `Collection.Indexed`s are always dense. "Unset"

\* indices and `undefined` indices are indistinguishable, and all indices from

\* 0 to `size` are visited when iterated.

\*

\* All Collection.Indexed methods return re-indexed Collections. In other words,

\* indices always start at 0 and increment until size. If you wish to

\* preserve indices, using them as keys, convert to a Collection.Keyed by

\* calling `toKeyedSeq`.

\*/

namespace Indexed {}

/\*\*

\* Creates a new Collection.Indexed.

\*

\* Note: `Collection.Indexed` is a conversion function and not a class, and

\* does not use the `new` keyword during construction.

\*/

function Indexed<T>(

collection?: Iterable<T> | ArrayLike<T>

): Collection.Indexed<T>;

interface Indexed<T> extends Collection<number, T> {

/\*\*

\* Deeply converts this Indexed collection to equivalent native JavaScript Array.

\*/

toJS(): Array<unknown>;

/\*\*

\* Shallowly converts this Indexed collection to equivalent native JavaScript Array.

\*/

toJSON(): Array<T>;

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<T>;

// Reading values

/\*\*

\* Returns the value associated with the provided index, or notSetValue if

\* the index is beyond the bounds of the Collection.

\*

\* `index` may be a negative number, which indexes back from the end of the

\* Collection. `s.get(-1)` gets the last item in the Collection.

\*/

get<NSV>(index: number, notSetValue: NSV): T | NSV;

get(index: number): T | undefined;

// Conversion to Seq

/\*\*

\* Returns Seq.Indexed.

\* @override

\*/

toSeq(): Seq.Indexed<T>;

/\*\*

\* If this is a collection of [key, value] entry tuples, it will return a

\* Seq.Keyed of those entries.

\*/

fromEntrySeq(): Seq.Keyed<unknown, unknown>;

// Combination

/\*\*

\* Returns a Collection of the same type with `separator` between each item

\* in this Collection.

\*/

interpose(separator: T): this;

/\*\*

\* Returns a Collection of the same type with the provided `collections`

\* interleaved into this collection.

\*

\* The resulting Collection includes the first item from each, then the

\* second from each, etc.

\*

\* <!-- runkit:activate

\* { "preamble": "require('immutable')"}

\* -->

\* ```js

\* const { List } = require('immutable')

\* List([ 1, 2, 3 ]).interleave(List([ 'A', 'B', 'C' ]))

\* // List [ 1, "A", 2, "B", 3, "C" ]

\* ```

\*

\* The shortest Collection stops interleave.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable')" }

\* -->

\* ```js

\* List([ 1, 2, 3 ]).interleave(

\* List([ 'A', 'B' ]),

\* List([ 'X', 'Y', 'Z' ])

\* )

\* // List [ 1, "A", "X", 2, "B", "Y" ]

\* ```

\*

\* Since `interleave()` re-indexes values, it produces a complete copy,

\* which has `O(N)` complexity.

\*

\* Note: `interleave` \*cannot\* be used in `withMutations`.

\*/

interleave(...collections: Array<Collection<unknown, T>>): this;

/\*\*

\* Splice returns a new indexed Collection by replacing a region of this

\* Collection with new values. If values are not provided, it only skips the

\* region to be removed.

\*

\* `index` may be a negative number, which indexes back from the end of the

\* Collection. `s.splice(-2)` splices after the second to last item.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* List([ 'a', 'b', 'c', 'd' ]).splice(1, 2, 'q', 'r', 's')

\* // List [ "a", "q", "r", "s", "d" ]

\* ```

\*

\* Since `splice()` re-indexes values, it produces a complete copy, which

\* has `O(N)` complexity.

\*

\* Note: `splice` \*cannot\* be used in `withMutations`.

\*/

splice(index: number, removeNum: number, ...values: Array<T>): this;

/\*\*

\* Returns a Collection of the same type "zipped" with the provided

\* collections.

\*

\* Like `zipWith`, but using the default `zipper`: creating an `Array`.

\*

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable')" }

\* -->

\* ```js

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 4, 5, 6 ]);

\* const c = a.zip(b); // List [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

\* ```

\*/

zip<U>(other: Collection<unknown, U>): Collection.Indexed<[T, U]>;

zip<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Collection.Indexed<[T, U, V]>;

zip(

...collections: Array<Collection<unknown, unknown>>

): Collection.Indexed<unknown>;

/\*\*

\* Returns a Collection "zipped" with the provided collections.

\*

\* Unlike `zip`, `zipAll` continues zipping until the longest collection is

\* exhausted. Missing values from shorter collections are filled with `undefined`.

\*

\* ```js

\* const a = List([ 1, 2 ]);

\* const b = List([ 3, 4, 5 ]);

\* const c = a.zipAll(b); // List [ [ 1, 3 ], [ 2, 4 ], [ undefined, 5 ] ]

\* ```

\*/

zipAll<U>(other: Collection<unknown, U>): Collection.Indexed<[T, U]>;

zipAll<U, V>(

other: Collection<unknown, U>,

other2: Collection<unknown, V>

): Collection.Indexed<[T, U, V]>;

zipAll(

...collections: Array<Collection<unknown, unknown>>

): Collection.Indexed<unknown>;

/\*\*

\* Returns a Collection of the same type "zipped" with the provided

\* collections by using a custom `zipper` function.

\*

\* <!-- runkit:activate

\* { "preamble": "const { List } = require('immutable')" }

\* -->

\* ```js

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 4, 5, 6 ]);

\* const c = a.zipWith((a, b) => a + b, b);

\* // List [ 5, 7, 9 ]

\* ```

\*/

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<unknown, U>

): Collection.Indexed<Z>;

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<unknown, U>,

thirdCollection: Collection<unknown, V>

): Collection.Indexed<Z>;

zipWith<Z>(

zipper: (...values: Array<unknown>) => Z,

...collections: Array<Collection<unknown, unknown>>

): Collection.Indexed<Z>;

// Search for value

/\*\*

\* Returns the first index at which a given value can be found in the

\* Collection, or -1 if it is not present.

\*/

indexOf(searchValue: T): number;

/\*\*

\* Returns the last index at which a given value can be found in the

\* Collection, or -1 if it is not present.

\*/

lastIndexOf(searchValue: T): number;

/\*\*

\* Returns the first index in the Collection where a value satisfies the

\* provided predicate function. Otherwise -1 is returned.

\*/

findIndex(

predicate: (value: T, index: number, iter: this) => boolean,

context?: unknown

): number;

/\*\*

\* Returns the last index in the Collection where a value satisfies the

\* provided predicate function. Otherwise -1 is returned.

\*/

findLastIndex(

predicate: (value: T, index: number, iter: this) => boolean,

context?: unknown

): number;

// Sequence algorithms

/\*\*

\* Returns a new Collection with other collections concatenated to this one.

\*/

concat<C>(

...valuesOrCollections: Array<Iterable<C> | C>

): Collection.Indexed<T | C>;

/\*\*

\* Returns a new Collection.Indexed with values passed through a

\* `mapper` function.

\*

\* ```js

\* const { Collection } = require('immutable')

\* Collection.Indexed([1,2]).map(x => 10 \* x)

\* // Seq [ 1, 2 ]

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: T, key: number, iter: this) => M,

context?: unknown

): Collection.Indexed<M>;

/\*\*

\* Flat-maps the Collection, returning a Collection of the same type.

\*

\* Similar to `collection.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: number, iter: this) => Iterable<M>,

context?: unknown

): Collection.Indexed<M>;

/\*\*

\* Returns a new Collection with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, index: number, iter: this) => value is F,

context?: unknown

): Collection.Indexed<F>;

filter(

predicate: (value: T, index: number, iter: this) => unknown,

context?: unknown

): this;

[Symbol.iterator](): IterableIterator<T>;

}

/\*\*

\* Set Collections only represent values. They have no associated keys or

\* indices. Duplicate values are possible in the lazy `Seq.Set`s, however

\* the concrete `Set` Collection does not allow duplicate values.

\*

\* Collection methods on Collection.Set such as `map` and `forEach` will provide

\* the value as both the first and second arguments to the provided function.

\*

\* ```js

\* const { Collection } = require('immutable')

\* const seq = Collection.Set([ 'A', 'B', 'C' ])

\* // Seq { "A", "B", "C" }

\* seq.forEach((v, k) =>

\* assert.equal(v, k)

\* )

\* ```

\*/

namespace Set {}

/\*\*

\* Similar to `Collection()`, but always returns a Collection.Set.

\*

\* Note: `Collection.Set` is a factory function and not a class, and does

\* not use the `new` keyword during construction.

\*/

function Set<T>(collection?: Iterable<T> | ArrayLike<T>): Collection.Set<T>;

interface Set<T> extends Collection<T, T> {

/\*\*

\* Deeply converts this Set collection to equivalent native JavaScript Array.

\*/

toJS(): Array<unknown>;

/\*\*

\* Shallowly converts this Set collection to equivalent native JavaScript Array.

\*/

toJSON(): Array<T>;

/\*\*

\* Shallowly converts this collection to an Array.

\*/

toArray(): Array<T>;

/\*\*

\* Returns Seq.Set.

\* @override

\*/

toSeq(): Seq.Set<T>;

// Sequence algorithms

/\*\*

\* Returns a new Collection with other collections concatenated to this one.

\*/

concat<U>(...collections: Array<Iterable<U>>): Collection.Set<T | U>;

/\*\*

\* Returns a new Collection.Set with values passed through a

\* `mapper` function.

\*

\* ```

\* Collection.Set([ 1, 2 ]).map(x => 10 \* x)

\* // Seq { 1, 2 }

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the

\* same value at every step.

\*/

map<M>(

mapper: (value: T, key: T, iter: this) => M,

context?: unknown

): Collection.Set<M>;

/\*\*

\* Flat-maps the Collection, returning a Collection of the same type.

\*

\* Similar to `collection.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: T, key: T, iter: this) => Iterable<M>,

context?: unknown

): Collection.Set<M>;

/\*\*

\* Returns a new Collection with only the values for which the `predicate`

\* function returns true.

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends T>(

predicate: (value: T, key: T, iter: this) => value is F,

context?: unknown

): Collection.Set<F>;

filter(

predicate: (value: T, key: T, iter: this) => unknown,

context?: unknown

): this;

[Symbol.iterator](): IterableIterator<T>;

}

}

/\*\*

\* Creates a Collection.

\*

\* The type of Collection created is based on the input.

\*

\* \* If an `Collection`, that same `Collection`.

\* \* If an Array-like, an `Collection.Indexed`.

\* \* If an Object with an Iterator defined, an `Collection.Indexed`.

\* \* If an Object, an `Collection.Keyed`.

\*

\* This methods forces the conversion of Objects and Strings to Collections.

\* If you want to ensure that a Collection of one item is returned, use

\* `Seq.of`.

\*

\* Note: An Iterator itself will be treated as an object, becoming a `Seq.Keyed`,

\* which is usually not what you want. You should turn your Iterator Object into

\* an iterable object by defining a Symbol.iterator (or @@iterator) method which

\* returns `this`.

\*

\* Note: `Collection` is a conversion function and not a class, and does not

\* use the `new` keyword during construction.

\*/

function Collection<I extends Collection<unknown, unknown>>(collection: I): I;

function Collection<T>(

collection: Iterable<T> | ArrayLike<T>

): Collection.Indexed<T>;

function Collection<V>(obj: {

[key: string]: V;

}): Collection.Keyed<string, V>;

function Collection<K = unknown, V = unknown>(): Collection<K, V>;

interface Collection<K, V> extends ValueObject {

// Value equality

/\*\*

\* True if this and the other Collection have value equality, as defined

\* by `Immutable.is()`.

\*

\* Note: This is equivalent to `Immutable.is(this, other)`, but provided to

\* allow for chained expressions.

\*/

equals(other: unknown): boolean;

/\*\*

\* Computes and returns the hashed identity for this Collection.

\*

\* The `hashCode` of a Collection is used to determine potential equality,

\* and is used when adding this to a `Set` or as a key in a `Map`, enabling

\* lookup via a different instance.

\*

\* <!-- runkit:activate

\* { "preamble": "const { Set, List } = require('immutable')" }

\* -->

\* ```js

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 1, 2, 3 ]);

\* assert.notStrictEqual(a, b); // different instances

\* const set = Set([ a ]);

\* assert.equal(set.has(b), true);

\* ```

\*

\* If two values have the same `hashCode`, they are [not guaranteed

\* to be equal][Hash Collision]. If two values have different `hashCode`s,

\* they must not be equal.

\*

\* [Hash Collision]: https://en.wikipedia.org/wiki/Collision\_(computer\_science)

\*/

hashCode(): number;

// Reading values

/\*\*

\* Returns the value associated with the provided key, or notSetValue if

\* the Collection does not contain this key.

\*

\* Note: it is possible a key may be associated with an `undefined` value,

\* so if `notSetValue` is not provided and this method returns `undefined`,

\* that does not guarantee the key was not found.

\*/

get<NSV>(key: K, notSetValue: NSV): V | NSV;

get(key: K): V | undefined;

/\*\*

\* True if a key exists within this `Collection`, using `Immutable.is`

\* to determine equality

\*/

has(key: K): boolean;

/\*\*

\* True if a value exists within this `Collection`, using `Immutable.is`

\* to determine equality

\* @alias contains

\*/

includes(value: V): boolean;

contains(value: V): boolean;

/\*\*

\* In case the `Collection` is not empty returns the first element of the

\* `Collection`.

\* In case the `Collection` is empty returns the optional default

\* value if provided, if no default value is provided returns undefined.

\*/

first<NSV = undefined>(notSetValue?: NSV): V | NSV;

/\*\*

\* In case the `Collection` is not empty returns the last element of the

\* `Collection`.

\* In case the `Collection` is empty returns the optional default

\* value if provided, if no default value is provided returns undefined.

\*/

last<NSV = undefined>(notSetValue?: NSV): V | NSV;

// Reading deep values

/\*\*

\* Returns the value found by following a path of keys or indices through

\* nested Collections.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, List } = require('immutable')

\* const deepData = Map({ x: List([ Map({ y: 123 }) ]) });

\* deepData.getIn(['x', 0, 'y']) // 123

\* ```

\*

\* Plain JavaScript Object or Arrays may be nested within an Immutable.js

\* Collection, and getIn() can access those values as well:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, List } = require('immutable')

\* const deepData = Map({ x: [ { y: 123 } ] });

\* deepData.getIn(['x', 0, 'y']) // 123

\* ```

\*/

getIn(searchKeyPath: Iterable<unknown>, notSetValue?: unknown): unknown;

/\*\*

\* True if the result of following a path of keys or indices through nested

\* Collections results in a set value.

\*/

hasIn(searchKeyPath: Iterable<unknown>): boolean;

// Persistent changes

/\*\*

\* This can be very useful as a way to "chain" a normal function into a

\* sequence of methods. RxJS calls this "let" and lodash calls it "thru".

\*

\* For example, to sum a Seq after mapping and filtering:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Seq } = require('immutable')

\*

\* function sum(collection) {

\* return collection.reduce((sum, x) => sum + x, 0)

\* }

\*

\* Seq([ 1, 2, 3 ])

\* .map(x => x + 1)

\* .filter(x => x % 2 === 0)

\* .update(sum)

\* // 6

\* ```

\*/

update<R>(updater: (value: this) => R): R;

// Conversion to JavaScript types

/\*\*

\* Deeply converts this Collection to equivalent native JavaScript Array or Object.

\*

\* `Collection.Indexed`, and `Collection.Set` become `Array`, while

\* `Collection.Keyed` become `Object`, converting keys to Strings.

\*/

toJS(): Array<unknown> | { [key: string]: unknown };

/\*\*

\* Shallowly converts this Collection to equivalent native JavaScript Array or Object.

\*

\* `Collection.Indexed`, and `Collection.Set` become `Array`, while

\* `Collection.Keyed` become `Object`, converting keys to Strings.

\*/

toJSON(): Array<V> | { [key: string]: V };

/\*\*

\* Shallowly converts this collection to an Array.

\*

\* `Collection.Indexed`, and `Collection.Set` produce an Array of values.

\* `Collection.Keyed` produce an Array of [key, value] tuples.

\*/

toArray(): Array<V> | Array<[K, V]>;

/\*\*

\* Shallowly converts this Collection to an Object.

\*

\* Converts keys to Strings.

\*/

toObject(): { [key: string]: V };

// Conversion to Collections

/\*\*

\* Converts this Collection to a Map, Throws if keys are not hashable.

\*

\* Note: This is equivalent to `Map(this.toKeyedSeq())`, but provided

\* for convenience and to allow for chained expressions.

\*/

toMap(): Map<K, V>;

/\*\*

\* Converts this Collection to a Map, maintaining the order of iteration.

\*

\* Note: This is equivalent to `OrderedMap(this.toKeyedSeq())`, but

\* provided for convenience and to allow for chained expressions.

\*/

toOrderedMap(): OrderedMap<K, V>;

/\*\*

\* Converts this Collection to a Set, discarding keys. Throws if values

\* are not hashable.

\*

\* Note: This is equivalent to `Set(this)`, but provided to allow for

\* chained expressions.

\*/

toSet(): Set<V>;

/\*\*

\* Converts this Collection to a Set, maintaining the order of iteration and

\* discarding keys.

\*

\* Note: This is equivalent to `OrderedSet(this.valueSeq())`, but provided

\* for convenience and to allow for chained expressions.

\*/

toOrderedSet(): OrderedSet<V>;

/\*\*

\* Converts this Collection to a List, discarding keys.

\*

\* This is similar to `List(collection)`, but provided to allow for chained

\* expressions. However, when called on `Map` or other keyed collections,

\* `collection.toList()` discards the keys and creates a list of only the

\* values, whereas `List(collection)` creates a list of entry tuples.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, List } = require('immutable')

\* var myMap = Map({ a: 'Apple', b: 'Banana' })

\* List(myMap) // List [ [ "a", "Apple" ], [ "b", "Banana" ] ]

\* myMap.toList() // List [ "Apple", "Banana" ]

\* ```

\*/

toList(): List<V>;

/\*\*

\* Converts this Collection to a Stack, discarding keys. Throws if values

\* are not hashable.

\*

\* Note: This is equivalent to `Stack(this)`, but provided to allow for

\* chained expressions.

\*/

toStack(): Stack<V>;

// Conversion to Seq

/\*\*

\* Converts this Collection to a Seq of the same kind (indexed,

\* keyed, or set).

\*/

toSeq(): Seq<K, V>;

/\*\*

\* Returns a Seq.Keyed from this Collection where indices are treated as keys.

\*

\* This is useful if you want to operate on an

\* Collection.Indexed and preserve the [index, value] pairs.

\*

\* The returned Seq will have identical iteration order as

\* this Collection.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Seq } = require('immutable')

\* const indexedSeq = Seq([ 'A', 'B', 'C' ])

\* // Seq [ "A", "B", "C" ]

\* indexedSeq.filter(v => v === 'B')

\* // Seq [ "B" ]

\* const keyedSeq = indexedSeq.toKeyedSeq()

\* // Seq { 0: "A", 1: "B", 2: "C" }

\* keyedSeq.filter(v => v === 'B')

\* // Seq { 1: "B" }

\* ```

\*/

toKeyedSeq(): Seq.Keyed<K, V>;

/\*\*

\* Returns an Seq.Indexed of the values of this Collection, discarding keys.

\*/

toIndexedSeq(): Seq.Indexed<V>;

/\*\*

\* Returns a Seq.Set of the values of this Collection, discarding keys.

\*/

toSetSeq(): Seq.Set<V>;

// Iterators

/\*\*

\* An iterator of this `Collection`'s keys.

\*

\* Note: this will return an ES6 iterator which does not support

\* Immutable.js sequence algorithms. Use `keySeq` instead, if this is

\* what you want.

\*/

keys(): IterableIterator<K>;

/\*\*

\* An iterator of this `Collection`'s values.

\*

\* Note: this will return an ES6 iterator which does not support

\* Immutable.js sequence algorithms. Use `valueSeq` instead, if this is

\* what you want.

\*/

values(): IterableIterator<V>;

/\*\*

\* An iterator of this `Collection`'s entries as `[ key, value ]` tuples.

\*

\* Note: this will return an ES6 iterator which does not support

\* Immutable.js sequence algorithms. Use `entrySeq` instead, if this is

\* what you want.

\*/

entries(): IterableIterator<[K, V]>;

[Symbol.iterator](): IterableIterator<unknown>;

// Collections (Seq)

/\*\*

\* Returns a new Seq.Indexed of the keys of this Collection,

\* discarding values.

\*/

keySeq(): Seq.Indexed<K>;

/\*\*

\* Returns an Seq.Indexed of the values of this Collection, discarding keys.

\*/

valueSeq(): Seq.Indexed<V>;

/\*\*

\* Returns a new Seq.Indexed of [key, value] tuples.

\*/

entrySeq(): Seq.Indexed<[K, V]>;

// Sequence algorithms

/\*\*

\* Returns a new Collection of the same type with values passed through a

\* `mapper` function.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Collection } = require('immutable')

\* Collection({ a: 1, b: 2 }).map(x => 10 \* x)

\* // Seq { "a": 10, "b": 20 }

\* ```

\*

\* Note: `map()` always returns a new instance, even if it produced the same

\* value at every step.

\*/

map<M>(

mapper: (value: V, key: K, iter: this) => M,

context?: unknown

): Collection<K, M>;

/\*\*

\* Note: used only for sets, which return Collection<M, M> but are otherwise

\* identical to normal `map()`.

\*

\* @ignore

\*/

map(...args: Array<never>): unknown;

/\*\*

\* Returns a new Collection of the same type with only the entries for which

\* the `predicate` function returns true.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ a: 1, b: 2, c: 3, d: 4}).filter(x => x % 2 === 0)

\* // Map { "b": 2, "d": 4 }

\* ```

\*

\* Note: `filter()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filter<F extends V>(

predicate: (value: V, key: K, iter: this) => value is F,

context?: unknown

): Collection<K, F>;

filter(

predicate: (value: V, key: K, iter: this) => unknown,

context?: unknown

): this;

/\*\*

\* Returns a new Collection of the same type with only the entries for which

\* the `predicate` function returns false.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ a: 1, b: 2, c: 3, d: 4}).filterNot(x => x % 2 === 0)

\* // Map { "a": 1, "c": 3 }

\* ```

\*

\* Note: `filterNot()` always returns a new instance, even if it results in

\* not filtering out any values.

\*/

filterNot(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): this;

/\*\*

\* Returns a new Collection of the same type in reverse order.

\*/

reverse(): this;

/\*\*

\* Returns a new Collection of the same type which includes the same entries,

\* stably sorted by using a `comparator`.

\*

\* If a `comparator` is not provided, a default comparator uses `<` and `>`.

\*

\* `comparator(valueA, valueB)`:

\*

\* \* Returns `0` if the elements should not be swapped.

\* \* Returns `-1` (or any negative number) if `valueA` comes before `valueB`

\* \* Returns `1` (or any positive number) if `valueA` comes after `valueB`

\* \* Is pure, i.e. it must always return the same value for the same pair

\* of values.

\*

\* When sorting collections which have no defined order, their ordered

\* equivalents will be returned. e.g. `map.sort()` returns OrderedMap.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* Map({ "c": 3, "a": 1, "b": 2 }).sort((a, b) => {

\* if (a < b) { return -1; }

\* if (a > b) { return 1; }

\* if (a === b) { return 0; }

\* });

\* // OrderedMap { "a": 1, "b": 2, "c": 3 }

\* ```

\*

\* Note: `sort()` Always returns a new instance, even if the original was

\* already sorted.

\*

\* Note: This is always an eager operation.

\*/

sort(comparator?: (valueA: V, valueB: V) => number): this;

/\*\*

\* Like `sort`, but also accepts a `comparatorValueMapper` which allows for

\* sorting by more sophisticated means:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* const beattles = Map({

\* John: { name: "Lennon" },

\* Paul: { name: "McCartney" },

\* George: { name: "Harrison" },

\* Ringo: { name: "Starr" },

\* });

\* beattles.sortBy(member => member.name);

\* ```

\*

\* Note: `sortBy()` Always returns a new instance, even if the original was

\* already sorted.

\*

\* Note: This is always an eager operation.

\*/

sortBy<C>(

comparatorValueMapper: (value: V, key: K, iter: this) => C,

comparator?: (valueA: C, valueB: C) => number

): this;

/\*\*

\* Returns a `Collection.Keyed` of `Collection.Keyeds`, grouped by the return

\* value of the `grouper` function.

\*

\* Note: This is always an eager operation.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List, Map } = require('immutable')

\* const listOfMaps = List([

\* Map({ v: 0 }),

\* Map({ v: 1 }),

\* Map({ v: 1 }),

\* Map({ v: 0 }),

\* Map({ v: 2 })

\* ])

\* const groupsOfMaps = listOfMaps.groupBy(x => x.get('v'))

\* // Map {

\* // 0: List [ Map{ "v": 0 }, Map { "v": 0 } ],

\* // 1: List [ Map{ "v": 1 }, Map { "v": 1 } ],

\* // 2: List [ Map{ "v": 2 } ],

\* // }

\* ```

\*/

groupBy<G>(

grouper: (value: V, key: K, iter: this) => G,

context?: unknown

): /\*Map\*/ Seq.Keyed<G, /\*this\*/ Collection<K, V>>;

// Side effects

/\*\*

\* The `sideEffect` is executed for every entry in the Collection.

\*

\* Unlike `Array#forEach`, if any call of `sideEffect` returns

\* `false`, the iteration will stop. Returns the number of entries iterated

\* (including the last iteration which returned false).

\*/

forEach(

sideEffect: (value: V, key: K, iter: this) => unknown,

context?: unknown

): number;

// Creating subsets

/\*\*

\* Returns a new Collection of the same type representing a portion of this

\* Collection from start up to but not including end.

\*

\* If begin is negative, it is offset from the end of the Collection. e.g.

\* `slice(-2)` returns a Collection of the last two entries. If it is not

\* provided the new Collection will begin at the beginning of this Collection.

\*

\* If end is negative, it is offset from the end of the Collection. e.g.

\* `slice(0, -1)` returns a Collection of everything but the last entry. If

\* it is not provided, the new Collection will continue through the end of

\* this Collection.

\*

\* If the requested slice is equivalent to the current Collection, then it

\* will return itself.

\*/

slice(begin?: number, end?: number): this;

/\*\*

\* Returns a new Collection of the same type containing all entries except

\* the first.

\*/

rest(): this;

/\*\*

\* Returns a new Collection of the same type containing all entries except

\* the last.

\*/

butLast(): this;

/\*\*

\* Returns a new Collection of the same type which excludes the first `amount`

\* entries from this Collection.

\*/

skip(amount: number): this;

/\*\*

\* Returns a new Collection of the same type which excludes the last `amount`

\* entries from this Collection.

\*/

skipLast(amount: number): this;

/\*\*

\* Returns a new Collection of the same type which includes entries starting

\* from when `predicate` first returns false.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* List([ 'dog', 'frog', 'cat', 'hat', 'god' ])

\* .skipWhile(x => x.match(/g/))

\* // List [ "cat", "hat", "god" ]

\* ```

\*/

skipWhile(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): this;

/\*\*

\* Returns a new Collection of the same type which includes entries starting

\* from when `predicate` first returns true.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* List([ 'dog', 'frog', 'cat', 'hat', 'god' ])

\* .skipUntil(x => x.match(/hat/))

\* // List [ "hat", "god" ]

\* ```

\*/

skipUntil(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): this;

/\*\*

\* Returns a new Collection of the same type which includes the first `amount`

\* entries from this Collection.

\*/

take(amount: number): this;

/\*\*

\* Returns a new Collection of the same type which includes the last `amount`

\* entries from this Collection.

\*/

takeLast(amount: number): this;

/\*\*

\* Returns a new Collection of the same type which includes entries from this

\* Collection as long as the `predicate` returns true.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* List([ 'dog', 'frog', 'cat', 'hat', 'god' ])

\* .takeWhile(x => x.match(/o/))

\* // List [ "dog", "frog" ]

\* ```

\*/

takeWhile(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): this;

/\*\*

\* Returns a new Collection of the same type which includes entries from this

\* Collection as long as the `predicate` returns false.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List } = require('immutable')

\* List([ 'dog', 'frog', 'cat', 'hat', 'god' ])

\* .takeUntil(x => x.match(/at/))

\* // List [ "dog", "frog" ]

\* ```

\*/

takeUntil(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): this;

// Combination

/\*\*

\* Returns a new Collection of the same type with other values and

\* collection-like concatenated to this one.

\*

\* For Seqs, all entries will be present in the resulting Seq, even if they

\* have the same key.

\*/

concat(

...valuesOrCollections: Array<unknown>

): Collection<unknown, unknown>;

/\*\*

\* Flattens nested Collections.

\*

\* Will deeply flatten the Collection by default, returning a Collection of the

\* same type, but a `depth` can be provided in the form of a number or

\* boolean (where true means to shallowly flatten one level). A depth of 0

\* (or shallow: false) will deeply flatten.

\*

\* Flattens only others Collection, not Arrays or Objects.

\*

\* Note: `flatten(true)` operates on Collection<unknown, Collection<K, V>> and

\* returns Collection<K, V>

\*/

flatten(depth?: number): Collection<unknown, unknown>;

// tslint:disable-next-line unified-signatures

flatten(shallow?: boolean): Collection<unknown, unknown>;

/\*\*

\* Flat-maps the Collection, returning a Collection of the same type.

\*

\* Similar to `collection.map(...).flatten(true)`.

\*/

flatMap<M>(

mapper: (value: V, key: K, iter: this) => Iterable<M>,

context?: unknown

): Collection<K, M>;

/\*\*

\* Flat-maps the Collection, returning a Collection of the same type.

\*

\* Similar to `collection.map(...).flatten(true)`.

\* Used for Dictionaries only.

\*/

flatMap<KM, VM>(

mapper: (value: V, key: K, iter: this) => Iterable<[KM, VM]>,

context?: unknown

): Collection<KM, VM>;

// Reducing a value

/\*\*

\* Reduces the Collection to a value by calling the `reducer` for every entry

\* in the Collection and passing along the reduced value.

\*

\* If `initialReduction` is not provided, the first item in the

\* Collection will be used.

\*

\* @see `Array#reduce`.

\*/

reduce<R>(

reducer: (reduction: R, value: V, key: K, iter: this) => R,

initialReduction: R,

context?: unknown

): R;

reduce<R>(

reducer: (reduction: V | R, value: V, key: K, iter: this) => R

): R;

/\*\*

\* Reduces the Collection in reverse (from the right side).

\*

\* Note: Similar to this.reverse().reduce(), and provided for parity

\* with `Array#reduceRight`.

\*/

reduceRight<R>(

reducer: (reduction: R, value: V, key: K, iter: this) => R,

initialReduction: R,

context?: unknown

): R;

reduceRight<R>(

reducer: (reduction: V | R, value: V, key: K, iter: this) => R

): R;

/\*\*

\* True if `predicate` returns true for all entries in the Collection.

\*/

every(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): boolean;

/\*\*

\* True if `predicate` returns true for any entry in the Collection.

\*/

some(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): boolean;

/\*\*

\* Joins values together as a string, inserting a separator between each.

\* The default separator is `","`.

\*/

join(separator?: string): string;

/\*\*

\* Returns true if this Collection includes no values.

\*

\* For some lazy `Seq`, `isEmpty` might need to iterate to determine

\* emptiness. At most one iteration will occur.

\*/

isEmpty(): boolean;

/\*\*

\* Returns the size of this Collection.

\*

\* Regardless of if this Collection can describe its size lazily (some Seqs

\* cannot), this method will always return the correct size. E.g. it

\* evaluates a lazy `Seq` if necessary.

\*

\* If `predicate` is provided, then this returns the count of entries in the

\* Collection for which the `predicate` returns true.

\*/

count(): number;

count(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): number;

/\*\*

\* Returns a `Seq.Keyed` of counts, grouped by the return value of

\* the `grouper` function.

\*

\* Note: This is not a lazy operation.

\*/

countBy<G>(

grouper: (value: V, key: K, iter: this) => G,

context?: unknown

): Map<G, number>;

// Search for value

/\*\*

\* Returns the first value for which the `predicate` returns true.

\*/

find(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown,

notSetValue?: V

): V | undefined;

/\*\*

\* Returns the last value for which the `predicate` returns true.

\*

\* Note: `predicate` will be called for each entry in reverse.

\*/

findLast(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown,

notSetValue?: V

): V | undefined;

/\*\*

\* Returns the first [key, value] entry for which the `predicate` returns true.

\*/

findEntry(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown,

notSetValue?: V

): [K, V] | undefined;

/\*\*

\* Returns the last [key, value] entry for which the `predicate`

\* returns true.

\*

\* Note: `predicate` will be called for each entry in reverse.

\*/

findLastEntry(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown,

notSetValue?: V

): [K, V] | undefined;

/\*\*

\* Returns the key for which the `predicate` returns true.

\*/

findKey(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): K | undefined;

/\*\*

\* Returns the last key for which the `predicate` returns true.

\*

\* Note: `predicate` will be called for each entry in reverse.

\*/

findLastKey(

predicate: (value: V, key: K, iter: this) => boolean,

context?: unknown

): K | undefined;

/\*\*

\* Returns the key associated with the search value, or undefined.

\*/

keyOf(searchValue: V): K | undefined;

/\*\*

\* Returns the last key associated with the search value, or undefined.

\*/

lastKeyOf(searchValue: V): K | undefined;

/\*\*

\* Returns the maximum value in this collection. If any values are

\* comparatively equivalent, the first one found will be returned.

\*

\* The `comparator` is used in the same way as `Collection#sort`. If it is not

\* provided, the default comparator is `>`.

\*

\* When two values are considered equivalent, the first encountered will be

\* returned. Otherwise, `max` will operate independent of the order of input

\* as long as the comparator is commutative. The default comparator `>` is

\* commutative \*only\* when types do not differ.

\*

\* If `comparator` returns 0 and either value is NaN, undefined, or null,

\* that value will be returned.

\*/

max(comparator?: (valueA: V, valueB: V) => number): V | undefined;

/\*\*

\* Like `max`, but also accepts a `comparatorValueMapper` which allows for

\* comparing by more sophisticated means:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List, } = require('immutable');

\* const l = List([

\* { name: 'Bob', avgHit: 1 },

\* { name: 'Max', avgHit: 3 },

\* { name: 'Lili', avgHit: 2 } ,

\* ]);

\* l.maxBy(i => i.avgHit); // will output { name: 'Max', avgHit: 3 }

\* ```

\*/

maxBy<C>(

comparatorValueMapper: (value: V, key: K, iter: this) => C,

comparator?: (valueA: C, valueB: C) => number

): V | undefined;

/\*\*

\* Returns the minimum value in this collection. If any values are

\* comparatively equivalent, the first one found will be returned.

\*

\* The `comparator` is used in the same way as `Collection#sort`. If it is not

\* provided, the default comparator is `<`.

\*

\* When two values are considered equivalent, the first encountered will be

\* returned. Otherwise, `min` will operate independent of the order of input

\* as long as the comparator is commutative. The default comparator `<` is

\* commutative \*only\* when types do not differ.

\*

\* If `comparator` returns 0 and either value is NaN, undefined, or null,

\* that value will be returned.

\*/

min(comparator?: (valueA: V, valueB: V) => number): V | undefined;

/\*\*

\* Like `min`, but also accepts a `comparatorValueMapper` which allows for

\* comparing by more sophisticated means:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List, } = require('immutable');

\* const l = List([

\* { name: 'Bob', avgHit: 1 },

\* { name: 'Max', avgHit: 3 },

\* { name: 'Lili', avgHit: 2 } ,

\* ]);

\* l.minBy(i => i.avgHit); // will output { name: 'Bob', avgHit: 1 }

\* ```

\*/

minBy<C>(

comparatorValueMapper: (value: V, key: K, iter: this) => C,

comparator?: (valueA: C, valueB: C) => number

): V | undefined;

// Comparison

/\*\*

\* True if `iter` includes every value in this Collection.

\*/

isSubset(iter: Iterable<V>): boolean;

/\*\*

\* True if this Collection includes every value in `iter`.

\*/

isSuperset(iter: Iterable<V>): boolean;

}

/\*\*

\* The interface to fulfill to qualify as a Value Object.

\*/

interface ValueObject {

/\*\*

\* True if this and the other Collection have value equality, as defined

\* by `Immutable.is()`.

\*

\* Note: This is equivalent to `Immutable.is(this, other)`, but provided to

\* allow for chained expressions.

\*/

equals(other: unknown): boolean;

/\*\*

\* Computes and returns the hashed identity for this Collection.

\*

\* The `hashCode` of a Collection is used to determine potential equality,

\* and is used when adding this to a `Set` or as a key in a `Map`, enabling

\* lookup via a different instance.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { List, Set } = require('immutable');

\* const a = List([ 1, 2, 3 ]);

\* const b = List([ 1, 2, 3 ]);

\* assert.notStrictEqual(a, b); // different instances

\* const set = Set([ a ]);

\* assert.equal(set.has(b), true);

\* ```

\*

\* Note: hashCode() MUST return a Uint32 number. The easiest way to

\* guarantee this is to return `myHash | 0` from a custom implementation.

\*

\* If two values have the same `hashCode`, they are [not guaranteed

\* to be equal][Hash Collision]. If two values have different `hashCode`s,

\* they must not be equal.

\*

\* Note: `hashCode()` is not guaranteed to always be called before

\* `equals()`. Most but not all Immutable.js collections use hash codes to

\* organize their internal data structures, while all Immutable.js

\* collections use equality during lookups.

\*

\* [Hash Collision]: https://en.wikipedia.org/wiki/Collision\_(computer\_science)

\*/

hashCode(): number;

}

/\*\*

\* Deeply converts plain JS objects and arrays to Immutable Maps and Lists.

\*

\* `fromJS` will convert Arrays and [array-like objects][2] to a List, and

\* plain objects (without a custom prototype) to a Map. [Iterable objects][3]

\* may be converted to List, Map, or Set.

\*

\* If a `reviver` is optionally provided, it will be called with every

\* collection as a Seq (beginning with the most nested collections

\* and proceeding to the top-level collection itself), along with the key

\* referring to each collection and the parent JS object provided as `this`.

\* For the top level, object, the key will be `""`. This `reviver` is expected

\* to return a new Immutable Collection, allowing for custom conversions from

\* deep JS objects. Finally, a `path` is provided which is the sequence of

\* keys to this value from the starting value.

\*

\* `reviver` acts similarly to the [same parameter in `JSON.parse`][1].

\*

\* If `reviver` is not provided, the default behavior will convert Objects

\* into Maps and Arrays into Lists like so:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { fromJS, isKeyed } = require('immutable')

\* function (key, value) {

\* return isKeyed(value) ? value.toMap() : value.toList()

\* }

\* ```

\*

\* Accordingly, this example converts native JS data to OrderedMap and List:

\*

\* <!-- runkit:activate -->

\* ```js

\* const { fromJS, isKeyed } = require('immutable')

\* fromJS({ a: {b: [10, 20, 30]}, c: 40}, function (key, value, path) {

\* console.log(key, value, path)

\* return isKeyed(value) ? value.toOrderedMap() : value.toList()

\* })

\*

\* > "b", [ 10, 20, 30 ], [ "a", "b" ]

\* > "a", {b: [10, 20, 30]}, [ "a" ]

\* > "", {a: {b: [10, 20, 30]}, c: 40}, []

\* ```

\*

\* Keep in mind, when using JS objects to construct Immutable Maps, that

\* JavaScript Object properties are always strings, even if written in a

\* quote-less shorthand, while Immutable Maps accept keys of any type.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map } = require('immutable')

\* let obj = { 1: "one" };

\* Object.keys(obj); // [ "1" ]

\* assert.equal(obj["1"], obj[1]); // "one" === "one"

\*

\* let map = Map(obj);

\* assert.notEqual(map.get("1"), map.get(1)); // "one" !== undefined

\* ```

\*

\* Property access for JavaScript Objects first converts the key to a string,

\* but since Immutable Map keys can be of any type the argument to `get()` is

\* not altered.

\*

\* [1]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/JSON/parse#Example.3A\_Using\_the\_reviver\_parameter

\* "Using the reviver parameter"

\* [2]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed\_collections#working\_with\_array-like\_objects

\* "Working with array-like objects"

\* [3]: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration\_protocols#the\_iterable\_protocol

\* "The iterable protocol"

\*/

function fromJS(

jsValue: unknown,

reviver?: (

key: string | number,

sequence: Collection.Keyed<string, unknown> | Collection.Indexed<unknown>,

path?: Array<string | number>

) => unknown

): Collection<unknown, unknown>;

/\*\*

\* Value equality check with semantics similar to `Object.is`, but treats

\* Immutable `Collection`s as values, equal if the second `Collection` includes

\* equivalent values.

\*

\* It's used throughout Immutable when checking for equality, including `Map`

\* key equality and `Set` membership.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { Map, is } = require('immutable')

\* const map1 = Map({ a: 1, b: 1, c: 1 })

\* const map2 = Map({ a: 1, b: 1, c: 1 })

\* assert.equal(map1 !== map2, true)

\* assert.equal(Object.is(map1, map2), false)

\* assert.equal(is(map1, map2), true)

\* ```

\*

\* `is()` compares primitive types like strings and numbers, Immutable.js

\* collections like `Map` and `List`, but also any custom object which

\* implements `ValueObject` by providing `equals()` and `hashCode()` methods.

\*

\* Note: Unlike `Object.is`, `Immutable.is` assumes `0` and `-0` are the same

\* value, matching the behavior of ES6 Map key equality.

\*/

function is(first: unknown, second: unknown): boolean;

/\*\*

\* The `hash()` function is an important part of how Immutable determines if

\* two values are equivalent and is used to determine how to store those

\* values. Provided with any value, `hash()` will return a 31-bit integer.

\*

\* When designing Objects which may be equal, it's important that when a

\* `.equals()` method returns true, that both values `.hashCode()` method

\* return the same value. `hash()` may be used to produce those values.

\*

\* For non-Immutable Objects that do not provide a `.hashCode()` functions

\* (including plain Objects, plain Arrays, Date objects, etc), a unique hash

\* value will be created for each \*instance\*. That is, the create hash

\* represents referential equality, and not value equality for Objects. This

\* ensures that if that Object is mutated over time that its hash code will

\* remain consistent, allowing Objects to be used as keys and values in

\* Immutable.js collections.

\*

\* Note that `hash()` attempts to balance between speed and avoiding

\* collisions, however it makes no attempt to produce secure hashes.

\*

\* \*New in Version 4.0\*

\*/

function hash(value: unknown): number;

/\*\*

\* True if `maybeImmutable` is an Immutable Collection or Record.

\*

\* Note: Still returns true even if the collections is within a `withMutations()`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isImmutable, Map, List, Stack } = require('immutable');

\* isImmutable([]); // false

\* isImmutable({}); // false

\* isImmutable(Map()); // true

\* isImmutable(List()); // true

\* isImmutable(Stack()); // true

\* isImmutable(Map().asMutable()); // true

\* ```

\*/

function isImmutable(

maybeImmutable: unknown

): maybeImmutable is Collection<unknown, unknown>;

/\*\*

\* True if `maybeCollection` is a Collection, or any of its subclasses.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isCollection, Map, List, Stack } = require('immutable');

\* isCollection([]); // false

\* isCollection({}); // false

\* isCollection(Map()); // true

\* isCollection(List()); // true

\* isCollection(Stack()); // true

\* ```

\*/

function isCollection(

maybeCollection: unknown

): maybeCollection is Collection<unknown, unknown>;

/\*\*

\* True if `maybeKeyed` is a Collection.Keyed, or any of its subclasses.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isKeyed, Map, List, Stack } = require('immutable');

\* isKeyed([]); // false

\* isKeyed({}); // false

\* isKeyed(Map()); // true

\* isKeyed(List()); // false

\* isKeyed(Stack()); // false

\* ```

\*/

function isKeyed(

maybeKeyed: unknown

): maybeKeyed is Collection.Keyed<unknown, unknown>;

/\*\*

\* True if `maybeIndexed` is a Collection.Indexed, or any of its subclasses.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isIndexed, Map, List, Stack, Set } = require('immutable');

\* isIndexed([]); // false

\* isIndexed({}); // false

\* isIndexed(Map()); // false

\* isIndexed(List()); // true

\* isIndexed(Stack()); // true

\* isIndexed(Set()); // false

\* ```

\*/

function isIndexed(

maybeIndexed: unknown

): maybeIndexed is Collection.Indexed<unknown>;

/\*\*

\* True if `maybeAssociative` is either a Keyed or Indexed Collection.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isAssociative, Map, List, Stack, Set } = require('immutable');

\* isAssociative([]); // false

\* isAssociative({}); // false

\* isAssociative(Map()); // true

\* isAssociative(List()); // true

\* isAssociative(Stack()); // true

\* isAssociative(Set()); // false

\* ```

\*/

function isAssociative(

maybeAssociative: unknown

): maybeAssociative is

| Collection.Keyed<unknown, unknown>

| Collection.Indexed<unknown>;

/\*\*

\* True if `maybeOrdered` is a Collection where iteration order is well

\* defined. True for Collection.Indexed as well as OrderedMap and OrderedSet.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { isOrdered, Map, OrderedMap, List, Set } = require('immutable');

\* isOrdered([]); // false

\* isOrdered({}); // false

\* isOrdered(Map()); // false

\* isOrdered(OrderedMap()); // true

\* isOrdered(List()); // true

\* isOrdered(Set()); // false

\* ```

\*/

function isOrdered(maybeOrdered: unknown): boolean;

/\*\*

\* True if `maybeValue` is a JavaScript Object which has \*both\* `equals()`

\* and `hashCode()` methods.

\*

\* Any two instances of \*value objects\* can be compared for value equality with

\* `Immutable.is()` and can be used as keys in a `Map` or members in a `Set`.

\*/

function isValueObject(maybeValue: unknown): maybeValue is ValueObject;

/\*\*

\* True if `maybeSeq` is a Seq.

\*/

function isSeq(

maybeSeq: unknown

): maybeSeq is

| Seq.Indexed<unknown>

| Seq.Keyed<unknown, unknown>

| Seq.Set<unknown>;

/\*\*

\* True if `maybeList` is a List.

\*/

function isList(maybeList: unknown): maybeList is List<unknown>;

/\*\*

\* True if `maybeMap` is a Map.

\*

\* Also true for OrderedMaps.

\*/

function isMap(maybeMap: unknown): maybeMap is Map<unknown, unknown>;

/\*\*

\* True if `maybeOrderedMap` is an OrderedMap.

\*/

function isOrderedMap(

maybeOrderedMap: unknown

): maybeOrderedMap is OrderedMap<unknown, unknown>;

/\*\*

\* True if `maybeStack` is a Stack.

\*/

function isStack(maybeStack: unknown): maybeStack is Stack<unknown>;

/\*\*

\* True if `maybeSet` is a Set.

\*

\* Also true for OrderedSets.

\*/

function isSet(maybeSet: unknown): maybeSet is Set<unknown>;

/\*\*

\* True if `maybeOrderedSet` is an OrderedSet.

\*/

function isOrderedSet(

maybeOrderedSet: unknown

): maybeOrderedSet is OrderedSet<unknown>;

/\*\*

\* True if `maybeRecord` is a Record.

\*/

function isRecord(maybeRecord: unknown): maybeRecord is Record<{}>;

/\*\*

\* Returns the value within the provided collection associated with the

\* provided key, or notSetValue if the key is not defined in the collection.

\*

\* A functional alternative to `collection.get(key)` which will also work on

\* plain Objects and Arrays as an alternative for `collection[key]`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { get } = require('immutable')

\* get([ 'dog', 'frog', 'cat' ], 2) // 'frog'

\* get({ x: 123, y: 456 }, 'x') // 123

\* get({ x: 123, y: 456 }, 'z', 'ifNotSet') // 'ifNotSet'

\* ```

\*/

function get<K, V>(collection: Collection<K, V>, key: K): V | undefined;

function get<K, V, NSV>(

collection: Collection<K, V>,

key: K,

notSetValue: NSV

): V | NSV;

function get<TProps extends object, K extends keyof TProps>(

record: Record<TProps>,

key: K,

notSetValue: unknown

): TProps[K];

function get<V>(collection: Array<V>, key: number): V | undefined;

function get<V, NSV>(

collection: Array<V>,

key: number,

notSetValue: NSV

): V | NSV;

function get<C extends object, K extends keyof C>(

object: C,

key: K,

notSetValue: unknown

): C[K];

function get<V>(collection: { [key: string]: V }, key: string): V | undefined;

function get<V, NSV>(

collection: { [key: string]: V },

key: string,

notSetValue: NSV

): V | NSV;

/\*\*

\* Returns true if the key is defined in the provided collection.

\*

\* A functional alternative to `collection.has(key)` which will also work with

\* plain Objects and Arrays as an alternative for

\* `collection.hasOwnProperty(key)`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { has } = require('immutable')

\* has([ 'dog', 'frog', 'cat' ], 2) // true

\* has([ 'dog', 'frog', 'cat' ], 5) // false

\* has({ x: 123, y: 456 }, 'x') // true

\* has({ x: 123, y: 456 }, 'z') // false

\* ```

\*/

function has(collection: object, key: unknown): boolean;

/\*\*

\* Returns a copy of the collection with the value at key removed.

\*

\* A functional alternative to `collection.remove(key)` which will also work

\* with plain Objects and Arrays as an alternative for

\* `delete collectionCopy[key]`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { remove } = require('immutable')

\* const originalArray = [ 'dog', 'frog', 'cat' ]

\* remove(originalArray, 1) // [ 'dog', 'cat' ]

\* console.log(originalArray) // [ 'dog', 'frog', 'cat' ]

\* const originalObject = { x: 123, y: 456 }

\* remove(originalObject, 'x') // { y: 456 }

\* console.log(originalObject) // { x: 123, y: 456 }

\* ```

\*/

function remove<K, C extends Collection<K, unknown>>(

collection: C,

key: K

): C;

function remove<

TProps extends object,

C extends Record<TProps>,

K extends keyof TProps

>(collection: C, key: K): C;

function remove<C extends Array<unknown>>(collection: C, key: number): C;

function remove<C, K extends keyof C>(collection: C, key: K): C;

function remove<C extends { [key: string]: unknown }, K extends keyof C>(

collection: C,

key: K

): C;

/\*\*

\* Returns a copy of the collection with the value at key set to the provided

\* value.

\*

\* A functional alternative to `collection.set(key, value)` which will also

\* work with plain Objects and Arrays as an alternative for

\* `collectionCopy[key] = value`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { set } = require('immutable')

\* const originalArray = [ 'dog', 'frog', 'cat' ]

\* set(originalArray, 1, 'cow') // [ 'dog', 'cow', 'cat' ]

\* console.log(originalArray) // [ 'dog', 'frog', 'cat' ]

\* const originalObject = { x: 123, y: 456 }

\* set(originalObject, 'x', 789) // { x: 789, y: 456 }

\* console.log(originalObject) // { x: 123, y: 456 }

\* ```

\*/

function set<K, V, C extends Collection<K, V>>(

collection: C,

key: K,

value: V

): C;

function set<

TProps extends object,

C extends Record<TProps>,

K extends keyof TProps

>(record: C, key: K, value: TProps[K]): C;

function set<V, C extends Array<V>>(collection: C, key: number, value: V): C;

function set<C, K extends keyof C>(object: C, key: K, value: C[K]): C;

function set<V, C extends { [key: string]: V }>(

collection: C,

key: string,

value: V

): C;

/\*\*

\* Returns a copy of the collection with the value at key set to the result of

\* providing the existing value to the updating function.

\*

\* A functional alternative to `collection.update(key, fn)` which will also

\* work with plain Objects and Arrays as an alternative for

\* `collectionCopy[key] = fn(collection[key])`.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { update } = require('immutable')

\* const originalArray = [ 'dog', 'frog', 'cat' ]

\* update(originalArray, 1, val => val.toUpperCase()) // [ 'dog', 'FROG', 'cat' ]

\* console.log(originalArray) // [ 'dog', 'frog', 'cat' ]

\* const originalObject = { x: 123, y: 456 }

\* update(originalObject, 'x', val => val \* 6) // { x: 738, y: 456 }

\* console.log(originalObject) // { x: 123, y: 456 }

\* ```

\*/

function update<K, V, C extends Collection<K, V>>(

collection: C,

key: K,

updater: (value: V | undefined) => V

): C;

function update<K, V, C extends Collection<K, V>, NSV>(

collection: C,

key: K,

notSetValue: NSV,

updater: (value: V | NSV) => V

): C;

function update<

TProps extends object,

C extends Record<TProps>,

K extends keyof TProps

>(record: C, key: K, updater: (value: TProps[K]) => TProps[K]): C;

function update<

TProps extends object,

C extends Record<TProps>,

K extends keyof TProps,

NSV

>(

record: C,

key: K,

notSetValue: NSV,

updater: (value: TProps[K] | NSV) => TProps[K]

): C;

function update<V>(

collection: Array<V>,

key: number,

updater: (value: V) => V

): Array<V>;

function update<V, NSV>(

collection: Array<V>,

key: number,

notSetValue: NSV,

updater: (value: V | NSV) => V

): Array<V>;

function update<C, K extends keyof C>(

object: C,

key: K,

updater: (value: C[K]) => C[K]

): C;

function update<C, K extends keyof C, NSV>(

object: C,

key: K,

notSetValue: NSV,

updater: (value: C[K] | NSV) => C[K]

): C;

function update<V, C extends { [key: string]: V }, K extends keyof C>(

collection: C,

key: K,

updater: (value: V) => V

): { [key: string]: V };

function update<V, C extends { [key: string]: V }, K extends keyof C, NSV>(

collection: C,

key: K,

notSetValue: NSV,

updater: (value: V | NSV) => V

): { [key: string]: V };

/\*\*

\* Returns the value at the provided key path starting at the provided

\* collection, or notSetValue if the key path is not defined.

\*

\* A functional alternative to `collection.getIn(keypath)` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { getIn } = require('immutable')

\* getIn({ x: { y: { z: 123 }}}, ['x', 'y', 'z']) // 123

\* getIn({ x: { y: { z: 123 }}}, ['x', 'q', 'p'], 'ifNotSet') // 'ifNotSet'

\* ```

\*/

function getIn(

collection: unknown,

keyPath: Iterable<unknown>,

notSetValue?: unknown

): unknown;

/\*\*

\* Returns true if the key path is defined in the provided collection.

\*

\* A functional alternative to `collection.hasIn(keypath)` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { hasIn } = require('immutable')

\* hasIn({ x: { y: { z: 123 }}}, ['x', 'y', 'z']) // true

\* hasIn({ x: { y: { z: 123 }}}, ['x', 'q', 'p']) // false

\* ```

\*/

function hasIn(collection: unknown, keyPath: Iterable<unknown>): boolean;

/\*\*

\* Returns a copy of the collection with the value at the key path removed.

\*

\* A functional alternative to `collection.removeIn(keypath)` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { removeIn } = require('immutable')

\* const original = { x: { y: { z: 123 }}}

\* removeIn(original, ['x', 'y', 'z']) // { x: { y: {}}}

\* console.log(original) // { x: { y: { z: 123 }}}

\* ```

\*/

function removeIn<C>(collection: C, keyPath: Iterable<unknown>): C;

/\*\*

\* Returns a copy of the collection with the value at the key path set to the

\* provided value.

\*

\* A functional alternative to `collection.setIn(keypath)` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { setIn } = require('immutable')

\* const original = { x: { y: { z: 123 }}}

\* setIn(original, ['x', 'y', 'z'], 456) // { x: { y: { z: 456 }}}

\* console.log(original) // { x: { y: { z: 123 }}}

\* ```

\*/

function setIn<C>(

collection: C,

keyPath: Iterable<unknown>,

value: unknown

): C;

/\*\*

\* Returns a copy of the collection with the value at key path set to the

\* result of providing the existing value to the updating function.

\*

\* A functional alternative to `collection.updateIn(keypath)` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { updateIn } = require('immutable')

\* const original = { x: { y: { z: 123 }}}

\* updateIn(original, ['x', 'y', 'z'], val => val \* 6) // { x: { y: { z: 738 }}}

\* console.log(original) // { x: { y: { z: 123 }}}

\* ```

\*/

function updateIn<C>(

collection: C,

keyPath: Iterable<unknown>,

updater: (value: unknown) => unknown

): C;

function updateIn<C>(

collection: C,

keyPath: Iterable<unknown>,

notSetValue: unknown,

updater: (value: unknown) => unknown

): C;

/\*\*

\* Returns a copy of the collection with the remaining collections merged in.

\*

\* A functional alternative to `collection.merge()` which will also work with

\* plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { merge } = require('immutable')

\* const original = { x: 123, y: 456 }

\* merge(original, { y: 789, z: 'abc' }) // { x: 123, y: 789, z: 'abc' }

\* console.log(original) // { x: 123, y: 456 }

\* ```

\*/

function merge<C>(

collection: C,

...collections: Array<

| Iterable<unknown>

| Iterable<[unknown, unknown]>

| { [key: string]: unknown }

>

): C;

/\*\*

\* Returns a copy of the collection with the remaining collections merged in,

\* calling the `merger` function whenever an existing value is encountered.

\*

\* A functional alternative to `collection.mergeWith()` which will also work

\* with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { mergeWith } = require('immutable')

\* const original = { x: 123, y: 456 }

\* mergeWith(

\* (oldVal, newVal) => oldVal + newVal,

\* original,

\* { y: 789, z: 'abc' }

\* ) // { x: 123, y: 1245, z: 'abc' }

\* console.log(original) // { x: 123, y: 456 }

\* ```

\*/

function mergeWith<C>(

merger: (oldVal: unknown, newVal: unknown, key: unknown) => unknown,

collection: C,

...collections: Array<

| Iterable<unknown>

| Iterable<[unknown, unknown]>

| { [key: string]: unknown }

>

): C;

/\*\*

\* Like `merge()`, but when two compatible collections are encountered with

\* the same key, it merges them as well, recursing deeply through the nested

\* data. Two collections are considered to be compatible (and thus will be

\* merged together) if they both fall into one of three categories: keyed

\* (e.g., `Map`s, `Record`s, and objects), indexed (e.g., `List`s and

\* arrays), or set-like (e.g., `Set`s). If they fall into separate

\* categories, `mergeDeep` will replace the existing collection with the

\* collection being merged in. This behavior can be customized by using

\* `mergeDeepWith()`.

\*

\* Note: Indexed and set-like collections are merged using

\* `concat()`/`union()` and therefore do not recurse.

\*

\* A functional alternative to `collection.mergeDeep()` which will also work

\* with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { mergeDeep } = require('immutable')

\* const original = { x: { y: 123 }}

\* mergeDeep(original, { x: { z: 456 }}) // { x: { y: 123, z: 456 }}

\* console.log(original) // { x: { y: 123 }}

\* ```

\*/

function mergeDeep<C>(

collection: C,

...collections: Array<

| Iterable<unknown>

| Iterable<[unknown, unknown]>

| { [key: string]: unknown }

>

): C;

/\*\*

\* Like `mergeDeep()`, but when two non-collections or incompatible

\* collections are encountered at the same key, it uses the `merger` function

\* to determine the resulting value. Collections are considered incompatible

\* if they fall into separate categories between keyed, indexed, and set-like.

\*

\* A functional alternative to `collection.mergeDeepWith()` which will also

\* work with plain Objects and Arrays.

\*

\* <!-- runkit:activate -->

\* ```js

\* const { mergeDeepWith } = require('immutable')

\* const original = { x: { y: 123 }}

\* mergeDeepWith(

\* (oldVal, newVal) => oldVal + newVal,

\* original,

\* { x: { y: 456 }}

\* ) // { x: { y: 579 }}

\* console.log(original) // { x: { y: 123 }}

\* ```

\*/

function mergeDeepWith<C>(

merger: (oldVal: unknown, newVal: unknown, key: unknown) => unknown,

collection: C,

...collections: Array<

| Iterable<unknown>

| Iterable<[unknown, unknown]>

| { [key: string]: unknown }

>

): C;

}

/\*\*

\* Defines the main export of the immutable module to be the Immutable namespace

\* This supports many common module import patterns:

\*

\* const Immutable = require("immutable");

\* const { List } = require("immutable");

\* import Immutable from "immutable";

\* import \* as Immutable from "immutable";

\* import { List } from "immutable";

\*

\*/

export = Immutable;

/\*\*

\* A global "Immutable" namespace used by UMD modules which allows the use of

\* the full Immutable API.

\*

\* If using Immutable as an imported module, prefer using:

\*

\* import Immutable from 'immutable'

\*

\*/

export as namespace Immutable;