/\*\*

\* Much of the Node.js core API is built around an idiomatic asynchronous

\* event-driven architecture in which certain kinds of objects (called "emitters")

\* emit named events that cause `Function` objects ("listeners") to be called.

\*

\* For instance: a `net.Server` object emits an event each time a peer

\* connects to it; a `fs.ReadStream` emits an event when the file is opened;

\* a `stream` emits an event whenever data is available to be read.

\*

\* All objects that emit events are instances of the `EventEmitter` class. These

\* objects expose an `eventEmitter.on()` function that allows one or more

\* functions to be attached to named events emitted by the object. Typically,

\* event names are camel-cased strings but any valid JavaScript property key

\* can be used.

\*

\* When the `EventEmitter` object emits an event, all of the functions attached

\* to that specific event are called \_synchronously\_. Any values returned by the

\* called listeners are \_ignored\_ and discarded.

\*

\* The following example shows a simple `EventEmitter` instance with a single

\* listener. The `eventEmitter.on()` method is used to register listeners, while

\* the `eventEmitter.emit()` method is used to trigger the event.

\*

\* ```js

\* const EventEmitter = require('events');

\*

\* class MyEmitter extends EventEmitter {}

\*

\* const myEmitter = new MyEmitter();

\* myEmitter.on('event', () => {

\* console.log('an event occurred!');

\* });

\* myEmitter.emit('event');

\* ```

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/events.js)

\*/

declare module 'events' {

interface EventEmitterOptions {

/\*\*

\* Enables automatic capturing of promise rejection.

\*/

captureRejections?: boolean | undefined;

}

interface NodeEventTarget {

once(eventName: string | symbol, listener: (...args: any[]) => void): this;

}

interface DOMEventTarget {

addEventListener(

eventName: string,

listener: (...args: any[]) => void,

opts?: {

once: boolean;

}

): any;

}

interface StaticEventEmitterOptions {

signal?: AbortSignal | undefined;

}

interface EventEmitter extends NodeJS.EventEmitter {}

/\*\*

\* The `EventEmitter` class is defined and exposed by the `events` module:

\*

\* ```js

\* const EventEmitter = require('events');

\* ```

\*

\* All `EventEmitter`s emit the event `'newListener'` when new listeners are

\* added and `'removeListener'` when existing listeners are removed.

\*

\* It supports the following option:

\* @since v0.1.26

\*/

class EventEmitter {

constructor(options?: EventEmitterOptions);

/\*\*

\* Creates a `Promise` that is fulfilled when the `EventEmitter` emits the given

\* event or that is rejected if the `EventEmitter` emits `'error'` while waiting.

\* The `Promise` will resolve with an array of all the arguments emitted to the

\* given event.

\*

\* This method is intentionally generic and works with the web platform [EventTarget](https://dom.spec.whatwg.org/#interface-eventtarget) interface, which has no special`'error'` event

\* semantics and does not listen to the `'error'` event.

\*

\* ```js

\* const { once, EventEmitter } = require('events');

\*

\* async function run() {

\* const ee = new EventEmitter();

\*

\* process.nextTick(() => {

\* ee.emit('myevent', 42);

\* });

\*

\* const [value] = await once(ee, 'myevent');

\* console.log(value);

\*

\* const err = new Error('kaboom');

\* process.nextTick(() => {

\* ee.emit('error', err);

\* });

\*

\* try {

\* await once(ee, 'myevent');

\* } catch (err) {

\* console.log('error happened', err);

\* }

\* }

\*

\* run();

\* ```

\*

\* The special handling of the `'error'` event is only used when `events.once()`is used to wait for another event. If `events.once()` is used to wait for the

\* '`error'` event itself, then it is treated as any other kind of event without

\* special handling:

\*

\* ```js

\* const { EventEmitter, once } = require('events');

\*

\* const ee = new EventEmitter();

\*

\* once(ee, 'error')

\* .then(([err]) => console.log('ok', err.message))

\* .catch((err) => console.log('error', err.message));

\*

\* ee.emit('error', new Error('boom'));

\*

\* // Prints: ok boom

\* ```

\*

\* An `AbortSignal` can be used to cancel waiting for the event:

\*

\* ```js

\* const { EventEmitter, once } = require('events');

\*

\* const ee = new EventEmitter();

\* const ac = new AbortController();

\*

\* async function foo(emitter, event, signal) {

\* try {

\* await once(emitter, event, { signal });

\* console.log('event emitted!');

\* } catch (error) {

\* if (error.name === 'AbortError') {

\* console.error('Waiting for the event was canceled!');

\* } else {

\* console.error('There was an error', error.message);

\* }

\* }

\* }

\*

\* foo(ee, 'foo', ac.signal);

\* ac.abort(); // Abort waiting for the event

\* ee.emit('foo'); // Prints: Waiting for the event was canceled!

\* ```

\* @since v11.13.0, v10.16.0

\*/

static once(emitter: NodeEventTarget, eventName: string | symbol, options?: StaticEventEmitterOptions): Promise<any[]>;

static once(emitter: DOMEventTarget, eventName: string, options?: StaticEventEmitterOptions): Promise<any[]>;

/\*\*

\* ```js

\* const { on, EventEmitter } = require('events');

\*

\* (async () => {

\* const ee = new EventEmitter();

\*

\* // Emit later on

\* process.nextTick(() => {

\* ee.emit('foo', 'bar');

\* ee.emit('foo', 42);

\* });

\*

\* for await (const event of on(ee, 'foo')) {

\* // The execution of this inner block is synchronous and it

\* // processes one event at a time (even with await). Do not use

\* // if concurrent execution is required.

\* console.log(event); // prints ['bar'] [42]

\* }

\* // Unreachable here

\* })();

\* ```

\*

\* Returns an `AsyncIterator` that iterates `eventName` events. It will throw

\* if the `EventEmitter` emits `'error'`. It removes all listeners when

\* exiting the loop. The `value` returned by each iteration is an array

\* composed of the emitted event arguments.

\*

\* An `AbortSignal` can be used to cancel waiting on events:

\*

\* ```js

\* const { on, EventEmitter } = require('events');

\* const ac = new AbortController();

\*

\* (async () => {

\* const ee = new EventEmitter();

\*

\* // Emit later on

\* process.nextTick(() => {

\* ee.emit('foo', 'bar');

\* ee.emit('foo', 42);

\* });

\*

\* for await (const event of on(ee, 'foo', { signal: ac.signal })) {

\* // The execution of this inner block is synchronous and it

\* // processes one event at a time (even with await). Do not use

\* // if concurrent execution is required.

\* console.log(event); // prints ['bar'] [42]

\* }

\* // Unreachable here

\* })();

\*

\* process.nextTick(() => ac.abort());

\* ```

\* @since v13.6.0, v12.16.0

\* @param eventName The name of the event being listened for

\* @return that iterates `eventName` events emitted by the `emitter`

\*/

static on(emitter: NodeJS.EventEmitter, eventName: string, options?: StaticEventEmitterOptions): AsyncIterableIterator<any>;

/\*\*

\* A class method that returns the number of listeners for the given `eventName`registered on the given `emitter`.

\*

\* ```js

\* const { EventEmitter, listenerCount } = require('events');

\* const myEmitter = new EventEmitter();

\* myEmitter.on('event', () => {});

\* myEmitter.on('event', () => {});

\* console.log(listenerCount(myEmitter, 'event'));

\* // Prints: 2

\* ```

\* @since v0.9.12

\* @deprecated Since v3.2.0 - Use `listenerCount` instead.

\* @param emitter The emitter to query

\* @param eventName The event name

\*/

static listenerCount(emitter: NodeJS.EventEmitter, eventName: string | symbol): number;

/\*\*

\* Returns a copy of the array of listeners for the event named `eventName`.

\*

\* For `EventEmitter`s this behaves exactly the same as calling `.listeners` on

\* the emitter.

\*

\* For `EventTarget`s this is the only way to get the event listeners for the

\* event target. This is useful for debugging and diagnostic purposes.

\*

\* ```js

\* const { getEventListeners, EventEmitter } = require('events');

\*

\* {

\* const ee = new EventEmitter();

\* const listener = () => console.log('Events are fun');

\* ee.on('foo', listener);

\* getEventListeners(ee, 'foo'); // [listener]

\* }

\* {

\* const et = new EventTarget();

\* const listener = () => console.log('Events are fun');

\* et.addEventListener('foo', listener);

\* getEventListeners(et, 'foo'); // [listener]

\* }

\* ```

\* @since v15.2.0, v14.17.0

\*/

static getEventListeners(emitter: DOMEventTarget | NodeJS.EventEmitter, name: string | symbol): Function[];

/\*\*

\* This symbol shall be used to install a listener for only monitoring `'error'`

\* events. Listeners installed using this symbol are called before the regular

\* `'error'` listeners are called.

\*

\* Installing a listener using this symbol does not change the behavior once an

\* `'error'` event is emitted, therefore the process will still crash if no

\* regular `'error'` listener is installed.

\*/

static readonly errorMonitor: unique symbol;

static readonly captureRejectionSymbol: unique symbol;

/\*\*

\* Sets or gets the default captureRejection value for all emitters.

\*/

// TODO: These should be described using static getter/setter pairs:

static captureRejections: boolean;

static defaultMaxListeners: number;

}

import internal = require('node:events');

namespace EventEmitter {

// Should just be `export { EventEmitter }`, but that doesn't work in TypeScript 3.4

export { internal as EventEmitter };

export interface Abortable {

/\*\*

\* When provided the corresponding `AbortController` can be used to cancel an asynchronous action.

\*/

signal?: AbortSignal | undefined;

}

}

global {

namespace NodeJS {

interface EventEmitter {

/\*\*

\* Alias for `emitter.on(eventName, listener)`.

\* @since v0.1.26

\*/

addListener(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Adds the `listener` function to the end of the listeners array for the

\* event named `eventName`. No checks are made to see if the `listener` has

\* already been added. Multiple calls passing the same combination of `eventName`and `listener` will result in the `listener` being added, and called, multiple

\* times.

\*

\* ```js

\* server.on('connection', (stream) => {

\* console.log('someone connected!');

\* });

\* ```

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\*

\* By default, event listeners are invoked in the order they are added. The`emitter.prependListener()` method can be used as an alternative to add the

\* event listener to the beginning of the listeners array.

\*

\* ```js

\* const myEE = new EventEmitter();

\* myEE.on('foo', () => console.log('a'));

\* myEE.prependListener('foo', () => console.log('b'));

\* myEE.emit('foo');

\* // Prints:

\* // b

\* // a

\* ```

\* @since v0.1.101

\* @param eventName The name of the event.

\* @param listener The callback function

\*/

on(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Adds a \*\*one-time\*\*`listener` function for the event named `eventName`. The

\* next time `eventName` is triggered, this listener is removed and then invoked.

\*

\* ```js

\* server.once('connection', (stream) => {

\* console.log('Ah, we have our first user!');

\* });

\* ```

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\*

\* By default, event listeners are invoked in the order they are added. The`emitter.prependOnceListener()` method can be used as an alternative to add the

\* event listener to the beginning of the listeners array.

\*

\* ```js

\* const myEE = new EventEmitter();

\* myEE.once('foo', () => console.log('a'));

\* myEE.prependOnceListener('foo', () => console.log('b'));

\* myEE.emit('foo');

\* // Prints:

\* // b

\* // a

\* ```

\* @since v0.3.0

\* @param eventName The name of the event.

\* @param listener The callback function

\*/

once(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Removes the specified `listener` from the listener array for the event named`eventName`.

\*

\* ```js

\* const callback = (stream) => {

\* console.log('someone connected!');

\* };

\* server.on('connection', callback);

\* // ...

\* server.removeListener('connection', callback);

\* ```

\*

\* `removeListener()` will remove, at most, one instance of a listener from the

\* listener array. If any single listener has been added multiple times to the

\* listener array for the specified `eventName`, then `removeListener()` must be

\* called multiple times to remove each instance.

\*

\* Once an event is emitted, all listeners attached to it at the

\* time of emitting are called in order. This implies that any`removeListener()` or `removeAllListeners()` calls \_after\_ emitting and\_before\_ the last listener finishes execution will

\* not remove them from`emit()` in progress. Subsequent events behave as expected.

\*

\* ```js

\* const myEmitter = new MyEmitter();

\*

\* const callbackA = () => {

\* console.log('A');

\* myEmitter.removeListener('event', callbackB);

\* };

\*

\* const callbackB = () => {

\* console.log('B');

\* };

\*

\* myEmitter.on('event', callbackA);

\*

\* myEmitter.on('event', callbackB);

\*

\* // callbackA removes listener callbackB but it will still be called.

\* // Internal listener array at time of emit [callbackA, callbackB]

\* myEmitter.emit('event');

\* // Prints:

\* // A

\* // B

\*

\* // callbackB is now removed.

\* // Internal listener array [callbackA]

\* myEmitter.emit('event');

\* // Prints:

\* // A

\* ```

\*

\* Because listeners are managed using an internal array, calling this will

\* change the position indices of any listener registered \_after\_ the listener

\* being removed. This will not impact the order in which listeners are called,

\* but it means that any copies of the listener array as returned by

\* the `emitter.listeners()` method will need to be recreated.

\*

\* When a single function has been added as a handler multiple times for a single

\* event (as in the example below), `removeListener()` will remove the most

\* recently added instance. In the example the `once('ping')`listener is removed:

\*

\* ```js

\* const ee = new EventEmitter();

\*

\* function pong() {

\* console.log('pong');

\* }

\*

\* ee.on('ping', pong);

\* ee.once('ping', pong);

\* ee.removeListener('ping', pong);

\*

\* ee.emit('ping');

\* ee.emit('ping');

\* ```

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\* @since v0.1.26

\*/

removeListener(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Alias for `emitter.removeListener()`.

\* @since v10.0.0

\*/

off(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Removes all listeners, or those of the specified `eventName`.

\*

\* It is bad practice to remove listeners added elsewhere in the code,

\* particularly when the `EventEmitter` instance was created by some other

\* component or module (e.g. sockets or file streams).

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\* @since v0.1.26

\*/

removeAllListeners(event?: string | symbol): this;

/\*\*

\* By default `EventEmitter`s will print a warning if more than `10` listeners are

\* added for a particular event. This is a useful default that helps finding

\* memory leaks. The `emitter.setMaxListeners()` method allows the limit to be

\* modified for this specific `EventEmitter` instance. The value can be set to`Infinity` (or `0`) to indicate an unlimited number of listeners.

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\* @since v0.3.5

\*/

setMaxListeners(n: number): this;

/\*\*

\* Returns the current max listener value for the `EventEmitter` which is either

\* set by `emitter.setMaxListeners(n)` or defaults to {@link defaultMaxListeners}.

\* @since v1.0.0

\*/

getMaxListeners(): number;

/\*\*

\* Returns a copy of the array of listeners for the event named `eventName`.

\*

\* ```js

\* server.on('connection', (stream) => {

\* console.log('someone connected!');

\* });

\* console.log(util.inspect(server.listeners('connection')));

\* // Prints: [ [Function] ]

\* ```

\* @since v0.1.26

\*/

listeners(eventName: string | symbol): Function[];

/\*\*

\* Returns a copy of the array of listeners for the event named `eventName`,

\* including any wrappers (such as those created by `.once()`).

\*

\* ```js

\* const emitter = new EventEmitter();

\* emitter.once('log', () => console.log('log once'));

\*

\* // Returns a new Array with a function `onceWrapper` which has a property

\* // `listener` which contains the original listener bound above

\* const listeners = emitter.rawListeners('log');

\* const logFnWrapper = listeners[0];

\*

\* // Logs "log once" to the console and does not unbind the `once` event

\* logFnWrapper.listener();

\*

\* // Logs "log once" to the console and removes the listener

\* logFnWrapper();

\*

\* emitter.on('log', () => console.log('log persistently'));

\* // Will return a new Array with a single function bound by `.on()` above

\* const newListeners = emitter.rawListeners('log');

\*

\* // Logs "log persistently" twice

\* newListeners[0]();

\* emitter.emit('log');

\* ```

\* @since v9.4.0

\*/

rawListeners(eventName: string | symbol): Function[];

/\*\*

\* Synchronously calls each of the listeners registered for the event named`eventName`, in the order they were registered, passing the supplied arguments

\* to each.

\*

\* Returns `true` if the event had listeners, `false` otherwise.

\*

\* ```js

\* const EventEmitter = require('events');

\* const myEmitter = new EventEmitter();

\*

\* // First listener

\* myEmitter.on('event', function firstListener() {

\* console.log('Helloooo! first listener');

\* });

\* // Second listener

\* myEmitter.on('event', function secondListener(arg1, arg2) {

\* console.log(`event with parameters ${arg1}, ${arg2} in second listener`);

\* });

\* // Third listener

\* myEmitter.on('event', function thirdListener(...args) {

\* const parameters = args.join(', ');

\* console.log(`event with parameters ${parameters} in third listener`);

\* });

\*

\* console.log(myEmitter.listeners('event'));

\*

\* myEmitter.emit('event', 1, 2, 3, 4, 5);

\*

\* // Prints:

\* // [

\* // [Function: firstListener],

\* // [Function: secondListener],

\* // [Function: thirdListener]

\* // ]

\* // Helloooo! first listener

\* // event with parameters 1, 2 in second listener

\* // event with parameters 1, 2, 3, 4, 5 in third listener

\* ```

\* @since v0.1.26

\*/

emit(eventName: string | symbol, ...args: any[]): boolean;

/\*\*

\* Returns the number of listeners listening to the event named `eventName`.

\* @since v3.2.0

\* @param eventName The name of the event being listened for

\*/

listenerCount(eventName: string | symbol): number;

/\*\*

\* Adds the `listener` function to the \_beginning\_ of the listeners array for the

\* event named `eventName`. No checks are made to see if the `listener` has

\* already been added. Multiple calls passing the same combination of `eventName`and `listener` will result in the `listener` being added, and called, multiple

\* times.

\*

\* ```js

\* server.prependListener('connection', (stream) => {

\* console.log('someone connected!');

\* });

\* ```

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\* @since v6.0.0

\* @param eventName The name of the event.

\* @param listener The callback function

\*/

prependListener(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Adds a \*\*one-time\*\*`listener` function for the event named `eventName` to the\_beginning\_ of the listeners array. The next time `eventName` is triggered, this

\* listener is removed, and then invoked.

\*

\* ```js

\* server.prependOnceListener('connection', (stream) => {

\* console.log('Ah, we have our first user!');

\* });

\* ```

\*

\* Returns a reference to the `EventEmitter`, so that calls can be chained.

\* @since v6.0.0

\* @param eventName The name of the event.

\* @param listener The callback function

\*/

prependOnceListener(eventName: string | symbol, listener: (...args: any[]) => void): this;

/\*\*

\* Returns an array listing the events for which the emitter has registered

\* listeners. The values in the array are strings or `Symbol`s.

\*

\* ```js

\* const EventEmitter = require('events');

\* const myEE = new EventEmitter();

\* myEE.on('foo', () => {});

\* myEE.on('bar', () => {});

\*

\* const sym = Symbol('symbol');

\* myEE.on(sym, () => {});

\*

\* console.log(myEE.eventNames());

\* // Prints: [ 'foo', 'bar', Symbol(symbol) ]

\* ```

\* @since v6.0.0

\*/

eventNames(): Array<string | symbol>;

}

}

}

export = EventEmitter;

}

declare module 'node:events' {

import events = require('events');

export = events;

}