Events

> *Stability: 2 - Stable*

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.

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Passing arguments and `this` to listeners

The `eventEmitter.emit()` method allows an arbitrary set of arguments to be passed to the listener functions. Keep in mind that when an ordinary listener function is called, the standard `this` keyword is intentionally set to reference the `EventEmitter` instance to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this, this === myEmitter);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined } true
});
myEmitter.emit('event', 'a', 'b');
```

It is possible to use ES6 Arrow Functions as listeners, however, when doing so, the `this` keyword will no longer reference the `EventEmitter` instance:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
  // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```

Asynchronous vs. synchronous

The `EventEmitter` calls all listeners synchronously in the order in which they were registered. This ensures the proper sequencing of events and helps avoid race conditions and logic errors. When appropriate, listener functions can switch to an asynchronous mode of operation using the `setImmediate()` or `process.nextTick()` methods:

```
const myEmitter = new MyEmitter();
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously');
  });
});
myEmitter.emit('event', 'a', 'b');
```

Handling events only once

When a listener is registered using the `eventEmitter.on()` method, that listener is invoked *every time* the named event is emitted.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.on('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
myEmitter.emit('event');
// Prints: 2
```

Using the `eventEmitter.once()` method, it is possible to register a listener that is called at most once for a particular event. Once the event is emitted, the listener is unregistered and *then* called.

```
const myEmitter = new MyEmitter();
let m = 0;
myEmitter.once('event', () => {
  console.log(++m);
});
myEmitter.emit('event');
// Prints: 1
```

```
myEmitter.emit('event');
// Ignored
```

Error events

When an error occurs within an `EventEmitter` instance, the typical action is for an `'error'` event to be emitted. These are treated as special cases within Node.js.

If an `EventEmitter` does *not* have at least one listener registered for the `'error'` event, and an `'error'` event is emitted, the error is thrown, a stack trace is printed, and the Node.js process exits.

```
const myEmitter = new MyEmitter();
myEmitter.emit('error', new Error('whoops!'));
// Throws and crashes Node.js
```

To guard against crashing the Node.js process the `domain` module can be used. (Note, however, that the `domain` module is deprecated.)

As a best practice, listeners should always be added for the `'error'` events.

```
const myEmitter = new MyEmitter();
myEmitter.on('error', (err) => {
  console.error('whoops! there was an error');
});
myEmitter.emit('error', new Error('whoops!'));
// Prints: whoops! there was an error
```

It is possible to monitor `'error'` events without consuming the emitted error by installing a listener using the symbol `events.errorMonitor`.

```
const { EventEmitter, errorMonitor } = require('events');

const myEmitter = new EventEmitter();
myEmitter.on(errorMonitor, (err) => {
  MyMonitoringTool.log(err);
});
myEmitter.emit('error', new Error('whoops!'));
// Still throws and crashes Node.js
```

Capture rejections of promises

Using `async` functions with event handlers is problematic, because it can lead to an unhandled rejection in case of a thrown exception:

```
const ee = new EventEmitter();
ee.on('something', async (value) => {
  throw new Error('kaboom');
});
```

The `captureRejections` option in the `EventEmitter` constructor or the global setting change this behavior, installing a `.then(undefined, handler)` handler on the `Promise`. This handler routes the exception asynchronously to the `Symbol.for('nodejs.rejection')` method if there is one, or to `'error'` event handler if there is none.

```
const ee1 = new EventEmitter({ captureRejections: true });
ee1.on('something', async (value) => {
  throw new Error('kaboom');
});

ee1.on('error', console.log);

const ee2 = new EventEmitter({ captureRejections: true });
ee2.on('something', async (value) => {
  throw new Error('kaboom');
});

ee2[Symbol.for('nodejs.rejection')] = console.log;
```

Setting `events.captureRejections = true` will change the default for all new instances of `EventEmitter`.

```
const events = require('events');
events.captureRejections = true;
const ee1 = new events.EventEmitter();
ee1.on('something', async (value) => {
  throw new Error('kaboom');
});

ee1.on('error', console.log);
```

The `'error'` events that are generated by the `captureRejections` behavior do not have a catch handler to avoid infinite error loops: the recommendation is to **not use `async` functions as `'error'` event handlers**.

Class: `EventEmitter`

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

```
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:

- `captureRejections` {boolean} It enables automatic capturing of promise rejection. **Default:** `false`.

Event: `'newListener'`

- `eventName` {string|symbol} The name of the event being listened for
- `listener` {Function} The event handler function

The `EventEmitter` instance will emit its own `'newListener'` event *before* a listener is added to its internal array of listeners.

Listeners registered for the `'newListener'` event are passed the event name and a reference to the listener being added.

The fact that the event is triggered before adding the listener has a subtle but important side effect: any *additional* listeners registered to the same `name` *within* the `'newListener'` callback are inserted *before* the listener that is in the process of being added.

```
class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
// Only do this once so we don't loop forever
myEmitter.once('newListener', (event, listener) => {
  if (event === 'event') {
    // Insert a new listener in front
    myEmitter.on('event', () => {
      console.log('B');
    });
  }
});
myEmitter.on('event', () => {
  console.log('A');
});
myEmitter.emit('event');
// Prints:
//   B
//   A
```

Event: `'removeListener'`

- `eventName` {string|symbol} The event name
- `listener` {Function} The event handler function

The `'removeListener'` event is emitted *after* the `listener` is removed.

`emitter.addListener(eventName, listener)`

- `eventName` {string|symbol}
- `listener` {Function}

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

`emitter.emit(eventName[, ...args])`

- `eventName` {string|symbol}
- `...args` {any}
- Returns: {boolean}

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.

```
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
```

`emitter.eventNames()`

- Returns: {Array}

Returns an array listing the events for which the emitter has registered listeners. The values in the array are strings or `Symbol` s.

```
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) ]
```

`emitter.getMaxListeners()`

- Returns: {integer}

Returns the current max listener value for the `EventEmitter` which is either set by `emitter.setMaxListeners(n)` or defaults to `events.defaultMaxListeners`.

`emitter.listenerCount(eventName)`

- `eventName` {string|symbol} The name of the event being listened for
- Returns: {integer}

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

`emitter.listeners(eventName)`

- `eventName` {string|symbol}
- Returns: {Function[]}

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

```
server.on('connection', (stream) => {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection')));
// Prints: [ [Function] ]
```

`emitter.off(eventName, listener)`

- `eventName` {string|symbol}
- `listener` {Function}
- Returns: {EventEmitter}

Alias for `emitter.removeListener()`.

`emitter.on(eventName, listener)`

- `eventName` {string|symbol} The name of the event.
- `listener` {Function} The callback function
- Returns: {EventEmitter}

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.

```
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.

```
const myEE = new EventEmitter();
myEE.on('foo', () => console.log('a'));
myEE.prependListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

`emitter.once(eventName, listener)`

- `eventName` {string|symbol} The name of the event.
- `listener` {Function} The callback function
- Returns: {EventEmitter}

Adds a **one-time** `listener` function for the event named `eventName`. The next time `eventName` is triggered, this listener is removed and then invoked.

```
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.

```
const myEE = new EventEmitter();
myEE.once('foo', () => console.log('a'));
myEE.prependOnceListener('foo', () => console.log('b'));
myEE.emit('foo');
// Prints:
//   b
//   a
```

`emitter.prependListener(eventName, listener)`

- `eventName` {string|symbol} The name of the event.
- `listener` {Function} The callback function
- Returns: {EventEmitter}

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.

```
server.prependListener('connection', (stream) => {
  console.log('someone connected!');
});
```

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

`emitter.prependOnceListener(eventName, listener)`

- `eventName` {string|symbol} The name of the event.
- `listener` {Function} The callback function
- Returns: {EventEmitter}

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.

```
server.prependOnceListener('connection', (stream) => {
  console.log('Ah, we have our first user!');
});
```

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

`emitter.removeAllListeners([eventName])`

- `eventName` {string|symbol}
- Returns: {EventEmitter}

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.

`emitter.removeListener(eventName, listener)`

- `eventName` {string|symbol}
- `listener` {Function}
- Returns: {EventEmitter}

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

```
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.

```
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:

```
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.

`emitter.setMaxListeners(n)`

- `n` {integer}
- Returns: {EventEmitter}

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.

`emitter.rawListeners(eventName)`

- `eventName` {string|symbol}
- Returns: {Function[]}

Returns a copy of the array of listeners for the event named `eventName`, including any wrappers (such as those created by `.once()`).

```
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');
```

`emitter[Symbol.for('nodejs.rejection')](err, eventName[, ...args])`

- `err` Error
- `eventName` {string|symbol}
- `...args` {any}

The `Symbol.for('nodejs.rejection')` method is called in case a promise rejection happens when emitting an event and `captureRejections` is enabled on the emitter. It is possible to use `events.captureRejectionSymbol` in place of `Symbol.for('nodejs.rejection')`.

```
const { EventEmitter, captureRejectionSymbol } = require('events');

class MyClass extends EventEmitter {
  constructor() {
    super({ captureRejections: true });
```

```
  }

  [captureRejectionSymbol](err, event, ...args) {
    console.log('rejection happened for', event, 'with', err, ...args);
    this.destroy(err);
  }

  destroy(err) {
    // Tear the resource down here.
  }
}
```

`events.defaultMaxListeners`

By default, a maximum of `10` listeners can be registered for any single event. This limit can be changed for individual `EventEmitter` instances using the `emitter.setMaxListeners(n)` method. To change the default for *all* `EventEmitter` instances, the `events.defaultMaxListeners` property can be used. If this value is not a positive number, a `RangeError` is thrown.

Take caution when setting the `events.defaultMaxListeners` because the change affects *all* `EventEmitter` instances, including those created before the change is made. However, calling `emitter.setMaxListeners(n)` still has precedence over `events.defaultMaxListeners`.

This is not a hard limit. The `EventEmitter` instance will allow more listeners to be added but will output a trace warning to stderr indicating that a "possible EventEmitter memory leak" has been detected. For any single `EventEmitter`, the `emitter.getMaxListeners()` and `emitter.setMaxListeners()` methods can be used to temporarily avoid this warning:

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', () => {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1, 0));
});
```

The `--trace-warnings` command-line flag can be used to display the stack trace for such warnings.

The emitted warning can be inspected with `process.on('warning')` and will have the additional `emitter`, `type` and `count` properties, referring to the event emitter instance, the event's name and the number of attached listeners, respectively. Its `name` property is set to `'MaxListenersExceededWarning'`.

`events.errorMonitor`

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.

`events.getEventListeners(emitterOrTarget, eventName)`

- `emitterOrTarget` {EventEmitter|EventTarget}
- `eventName` {string|symbol}
- Returns: {Function[]}

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.

```
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]
}
```

`events.once(emitter, name[, options])`

- `emitter` {EventEmitter}
- `name` {string}
- `options` {Object}
  - `signal` {AbortSignal} Can be used to cancel waiting for the event.
- Returns: {Promise}

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 interface, which has no special `'error'` event semantics and does not listen to the `'error'` event.

```
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:

```
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:

```
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!
```

Awaiting multiple events emitted on `process.nextTick()`

There is an edge case worth noting when using the `events.once()` function to await multiple events emitted on in the same batch of `process.nextTick()` operations, or whenever multiple events are emitted synchronously. Specifically, because the `process.nextTick()` queue is drained before the `Promise` microtask queue, and because `EventEmitter` emits all events synchronously, it is possible for `events.once()` to miss an event.

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

const myEE = new EventEmitter();

async function foo() {
  await once(myEE, 'bar');
  console.log('bar');

  // This Promise will never resolve because the 'foo' event will
  // have already been emitted before the Promise is created.
  await once(myEE, 'foo');
  console.log('foo');
}

process.nextTick(() => {
  myEE.emit('bar');
  myEE.emit('foo');
});

foo().then(() => console.log('done'));
```

To catch both events, create each of the Promises *before* awaiting either of them, then it becomes possible to use `Promise.all()`, `Promise.race()`, or `Promise.allSettled()`:

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

const myEE = new EventEmitter();

async function foo() {
  await Promise.all([once(myEE, 'bar'), once(myEE, 'foo')]);
  console.log('foo', 'bar');
}

process.nextTick(() => {
  myEE.emit('bar');
  myEE.emit('foo');
});

foo().then(() => console.log('done'));
```

`events.captureRejections`

Value: {boolean}

Change the default `captureRejections` option on all new `EventEmitter` objects.

`events.captureRejectionSymbol`

Value: `Symbol.for('nodejs.rejection')`

See how to write a custom rejection handler.

`events.listenerCount(emitter, eventName)`

> *Stability: 0 - Deprecated: Use `emitter.listenerCount()` instead.*

- `emitter` {EventEmitter} The emitter to query
- `eventName` {string|symbol} The event name

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

```
const { EventEmitter, listenerCount } = require('events');
const myEmitter = new EventEmitter();
myEmitter.on('event', () => {});
myEmitter.on('event', () => {});
console.log(listenerCount(myEmitter, 'event'));
// Prints: 2
```

`events.on(emitter, eventName[, options])`

- `emitter` {EventEmitter}
- `eventName` {string|symbol} The name of the event being listened for
- `options` {Object}
  - `signal` {AbortSignal} Can be used to cancel awaiting events.
- Returns: {AsyncIterator} that iterates `eventName` events emitted by the `emitter`

```
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:

```
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());
```

`events.setMaxListeners(n[, ...eventTargets])`

- `n` {number} A non-negative number. The maximum number of listeners per `EventTarget` event.
- `...eventsTargets` {EventTarget[]|EventEmitter[]} Zero or more {EventTarget} or {EventEmitter} instances. If none are specified, `n` is set as the default max for all newly created {EventTarget} and {EventEmitter} objects.

```
const {
  setMaxListeners,
  EventEmitter
} = require('events');

const target = new EventTarget();
```

```
const emitter = new EventEmitter();

setMaxListeners(5, target, emitter);
```

Class: `events.EventEmitterAsyncResource extends EventEmitter`

Integrates `EventEmitter` with {AsyncResource} for `EventEmitter` s that require manual async tracking. Specifically, all events emitted by instances of `events.EventEmitterAsyncResource` will run within its async context.

```
const { EventEmitterAsyncResource } = require('events');
const { notStrictEqual, strictEqual } = require('assert');
const { executionAsyncId } = require('async_hooks');

// Async tracking tooling will identify this as 'Q'.
const ee1 = new EventEmitterAsyncResource({ name: 'Q' });

// 'foo' listeners will run in the EventEmitters async context.
ee1.on('foo', () => {
  strictEqual(executionAsyncId(), ee1.asyncId);
  strictEqual(triggerAsyncId(), ee1.triggerAsyncId);
});

const ee2 = new EventEmitter();

// 'foo' listeners on ordinary EventEmitters that do not track async
// context, however, run in the same async context as the emit().
ee2.on('foo', () => {
  notStrictEqual(executionAsyncId(), ee2.asyncId);
  notStrictEqual(triggerAsyncId(), ee2.triggerAsyncId);
});

Promise.resolve().then(() => {
  ee1.emit('foo');
  ee2.emit('foo');
});
```

The `EventEmitterAsyncResource` class has the same methods and takes the same options as `EventEmitter` and `AsyncResource` themselves.

`new events.EventEmitterAsyncResource(options)`

- `options` {Object}
  - `captureRejections` {boolean} It enables automatic capturing of promise rejection. **Default:** `false` .
  - `name` {string} The type of async event. **Default::** `new.target.name` .
  - `triggerAsyncId` {number} The ID of the execution context that created this async event. **Default:** `executionAsyncId()` .
  - `requireManualDestroy` {boolean} If set to `true` , disables `emitDestroy` when the object is garbage collected. This usually does not need to be set (even if `emitDestroy` is called

manually), unless the resource's `asyncId` is retrieved and the sensitive API's `emitDestroy` is called with it. When set to `false`, the `emitDestroy` call on garbage collection will only take place if there is at least one active `destroy` hook. **Default:** `false`.

`eventemitterasyncresource.asyncId`

- Type: {number} The unique `asyncId` assigned to the resource.

`eventemitterasyncresource.asyncResource`

- Type: The underlying {AsyncResource}.

The returned `AsyncResource` object has an additional `eventEmitter` property that provides a reference to this `EventEmitterAsyncResource`.

`eventemitterasyncresource.emitDestroy()`

Call all `destroy` hooks. This should only ever be called once. An error will be thrown if it is called more than once. This **must** be manually called. If the resource is left to be collected by the GC then the `destroy` hooks will never be called.

`eventemitterasyncresource.triggerAsyncId`

- Type: {number} The same `triggerAsyncId` that is passed to the `AsyncResource` constructor.

`EventTarget` and `Event` API

The `EventTarget` and `Event` objects are a Node.js-specific implementation of the `EventTarget` Web API that are exposed by some Node.js core APIs.

```
const target = new EventTarget();

target.addEventListener('foo', (event) => {
  console.log('foo event happened!');
});
```

Node.js `EventTarget` vs. DOM `EventTarget`

There are two key differences between the Node.js `EventTarget` and the `EventTarget` Web API:

1. Whereas DOM `EventTarget` instances *may* be hierarchical, there is no concept of hierarchy and event propagation in Node.js. That is, an event dispatched to an `EventTarget` does not propagate through a hierarchy of nested target objects that may each have their own set of handlers for the event.
2. In the Node.js `EventTarget`, if an event listener is an async function or returns a `Promise`, and the returned `Promise` rejects, the rejection is automatically captured and handled the same way as a listener that throws synchronously (see `EventTarget` error handling for details).

`NodeEventTarget` vs. `EventEmitter`

The `NodeEventTarget` object implements a modified subset of the `EventEmitter` API that allows it to closely *emulate* an `EventEmitter` in certain situations. A `NodeEventTarget` is *not* an instance of `EventEmitter` and cannot be used in place of an `EventEmitter` in most cases.

1. Unlike `EventEmitter`, any given `listener` can be registered at most once per event `type`. Attempts to register a `listener` multiple times are ignored.
2. The `NodeEventTarget` does not emulate the full `EventEmitter` API. Specifically the `prependListener()`, `prependOnceListener()`, `rawListeners()`, `setMaxListeners()`, `getMaxListeners()`, and `errorMonitor` APIs are not emulated. The `'newListener'` and `'removeListener'` events will also not be emitted.
3. The `NodeEventTarget` does not implement any special default behavior for events with type `'error'`.
4. The `NodeEventTarget` supports `EventListener` objects as well as functions as handlers for all event types.

Event listener

Event listeners registered for an event `type` may either be JavaScript functions or objects with a `handleEvent` property whose value is a function.

In either case, the handler function is invoked with the `event` argument passed to the `eventTarget.dispatchEvent()` function.

Async functions may be used as event listeners. If an async handler function rejects, the rejection is captured and handled as described in `EventTarget` error handling.

An error thrown by one handler function does not prevent the other handlers from being invoked.

The return value of a handler function is ignored.

Handlers are always invoked in the order they were added.

Handler functions may mutate the `event` object.

```
function handler1(event) {
  console.log(event.type);  // Prints 'foo'
  event.a = 1;
}

async function handler2(event) {
  console.log(event.type);  // Prints 'foo'
  console.log(event.a);  // Prints 1
}

const handler3 = {
  handleEvent(event) {
    console.log(event.type);  // Prints 'foo'
  }
};

const handler4 = {
  async handleEvent(event) {
    console.log(event.type);  // Prints 'foo'
  }
};

const target = new EventTarget();
```

```
target.addEventListener('foo', handler1);
target.addEventListener('foo', handler2);
target.addEventListener('foo', handler3);
target.addEventListener('foo', handler4, { once: true });
```

`EventTarget` error handling

When a registered event listener throws (or returns a Promise that rejects), by default the error is treated as an uncaught exception on `process.nextTick()`. This means uncaught exceptions in `EventTarget`s will terminate the Node.js process by default.

Throwing within an event listener will *not* stop the other registered handlers from being invoked.

The `EventTarget` does not implement any special default handling for `'error'` type events like `EventEmitter`.

Currently errors are first forwarded to the `process.on('error')` event before reaching `process.on('uncaughtException')`. This behavior is deprecated and will change in a future release to align `EventTarget` with other Node.js APIs. Any code relying on the `process.on('error')` event should be aligned with the new behavior.

Class: `Event`

The `Event` object is an adaptation of the `Event` Web API. Instances are created internally by Node.js.

`event.bubbles`

- Type: {boolean} Always returns `false`.

This is not used in Node.js and is provided purely for completeness.

`event.cancelBubble()`

Alias for `event.stopPropagation()`. This is not used in Node.js and is provided purely for completeness.

`event.cancelable`

- Type: {boolean} True if the event was created with the `cancelable` option.

`event.composed`

- Type: {boolean} Always returns `false`.

This is not used in Node.js and is provided purely for completeness.

`event.composedPath()`

Returns an array containing the current `EventTarget` as the only entry or empty if the event is not being dispatched. This is not used in Node.js and is provided purely for completeness.

`event.currentTarget`

- Type: {EventTarget} The `EventTarget` dispatching the event.

Alias for `event.target`.

`event.defaultPrevented`

- Type: {boolean}

Is `true` if `cancelable` is `true` and `event.preventDefault()` has been called.

`event.eventPhase`

- Type: {number} Returns `0` while an event is not being dispatched, `2` while it is being dispatched.

This is not used in Node.js and is provided purely for completeness.

`event.isTrusted`

- Type: {boolean}

The {AbortSignal} `"abort"` event is emitted with `isTrusted` set to `true`. The value is `false` in all other cases.

`event.preventDefault()`

Sets the `defaultPrevented` property to `true` if `cancelable` is `true`.

`event.returnValue`

- Type: {boolean} True if the event has not been canceled.

This is not used in Node.js and is provided purely for completeness.

`event.srcElement`

- Type: {EventTarget} The `EventTarget` dispatching the event.

Alias for `event.target`.

`event.stopImmediatePropagation()`

Stops the invocation of event listeners after the current one completes.

`event.stopPropagation()`

This is not used in Node.js and is provided purely for completeness.

`event.target`

- Type: {EventTarget} The `EventTarget` dispatching the event.

`event.timeStamp`

- Type: {number}

The millisecond timestamp when the `Event` object was created.

`event.type`

- Type: {string}

The event type identifier.

Class: `EventTarget`

`eventTarget.addEventListener(type, listener[, options])`

- `type` {string}
- `listener` {Function|EventListener}

- `options` {Object}
  - `once` {boolean} When `true`, the listener is automatically removed when it is first invoked. **Default:** `false`.
  - `passive` {boolean} When `true`, serves as a hint that the listener will not call the `Event` object's `preventDefault()` method. **Default:** `false`.
  - `capture` {boolean} Not directly used by Node.js. Added for API completeness. **Default:** `false`.

Adds a new handler for the `type` event. Any given `listener` is added only once per `type` and per `capture` option value.

If the `once` option is `true`, the `listener` is removed after the next time a `type` event is dispatched.

The `capture` option is not used by Node.js in any functional way other than tracking registered event listeners per the `EventTarget` specification. Specifically, the `capture` option is used as part of the key when registering a `listener`. Any individual `listener` may be added once with `capture = false`, and once with `capture = true`.

```
function handler(event) {}

const target = new EventTarget();
target.addEventListener('foo', handler, { capture: true });  // first
target.addEventListener('foo', handler, { capture: false }); // second

// Removes the second instance of handler
target.removeEventListener('foo', handler);

// Removes the first instance of handler
target.removeEventListener('foo', handler, { capture: true });
```

`eventTarget.dispatchEvent(event)`

- `event` {Event}
- Returns: {boolean} `true` if either event's `cancelable` attribute value is false or its `preventDefault()` method was not invoked, otherwise `false`.

Dispatches the `event` to the list of handlers for `event.type`.

The registered event listeners is synchronously invoked in the order they were registered.

`eventTarget.removeEventListener(type, listener)`

- `type` {string}
- `listener` {Function|EventListener}
- `options` {Object}
  - `capture` {boolean}

Removes the `listener` from the list of handlers for event `type`.

Class: `NodeEventTarget`

- Extends: {EventTarget}

The `NodeEventTarget` is a Node.js-specific extension to `EventTarget` that emulates a subset of the `EventEmitter` API.

`nodeEventTarget.addListener(type, listener[, options])`

- `type` {string}
- `listener` {Function|EventListener}
- `options` {Object}
  - `once` {boolean}
- Returns: {EventTarget} this

Node.js-specific extension to the `EventTarget` class that emulates the equivalent `EventEmitter` API. The only difference between `addListener()` and `addEventListener()` is that `addListener()` will return a reference to the `EventTarget`.

`nodeEventTarget.eventNames()`

- Returns: {string[]}

Node.js-specific extension to the `EventTarget` class that returns an array of event `type` names for which event listeners are registered.

`nodeEventTarget.listenerCount(type)`

- `type` {string}
- Returns: {number}

Node.js-specific extension to the `EventTarget` class that returns the number of event listeners registered for the `type`.

`nodeEventTarget.off(type, listener)`

- `type` {string}
- `listener` {Function|EventListener}
- Returns: {EventTarget} this

Node.js-specific alias for `eventTarget.removeListener()`.

`nodeEventTarget.on(type, listener[, options])`

- `type` {string}
- `listener` {Function|EventListener}
- `options` {Object}
  - `once` {boolean}
- Returns: {EventTarget} this

Node.js-specific alias for `eventTarget.addListener()`.

`nodeEventTarget.once(type, listener[, options])`

- `type` {string}
- `listener` {Function|EventListener}
- `options` {Object}
- Returns: {EventTarget} this

Node.js-specific extension to the `EventTarget` class that adds a `once` listener for the given event `type`. This is equivalent to calling `on` with the `once` option set to `true`.

`nodeEventTarget.removeAllListeners([type])`

- `type` {string}
- Returns: {EventTarget} this

Node.js-specific extension to the `EventTarget` class. If `type` is specified, removes all registered listeners for `type`, otherwise removes all registered listeners.

`nodeEventTarget.removeListener(type, listener)`

- `type` {string}
- `listener` {Function|EventListener}
- Returns: {EventTarget} this

Node.js-specific extension to the `EventTarget` class that removes the `listener` for the given `type`. The only difference between `removeListener()` and `removeEventListener()` is that `removeListener()` will return a reference to the `EventTarget`.