Node.js EventEmitter

A Refresher on Events

- Events are actions that happen while a program is running
- We add "listeners" to events using functions known as "callbacks"
- These "callbacks" typically receive some data about the event that happened

```
// Prints the X and Y position when clicked
$("#element").on("click", function(event) {
  console.log("X: " + event.x + " | Y: " + event.y);
});
```

Events in Node

- In the browser, we've used events to listen to user interactions
- While there may be nothing for a user to `click` on or `mouseenter` over in Node, events still exist
- However, the events don't come from elements or the window
- Instead they come from `EventEmitters`

Examples of Node Events

- File I/O - open, `close` on files
- HTTP - `connect`, `continue`, `abort` on connections
- Error handling - `uncaughtException` on any emitter

Creating an EventEmitter

- `EventEmitter` is a class that comes from the `"events"` core module in node
- You can grab the class by `requireing` the `"events"` module.

```
// Grab the EventEmitter class from the "events" module
const EventEmitter = require("events");

// Create a new instance of EventEmitter
const emitter = new EventEmitter();
```

Listening with `.addListener()` / `.on()`

- Both `.addListener` and `.on` do the exact same thing
- Lets you listen for an event, just like jQuery's `.on`

```
const EventEmitter = require("events");
const emitter = new EventEmitter();

// Whenever `some-event` happens, print out the message it was sent with
emitter.on("some-event", function(message) {
  console.log(message);
});
```

Emitting with `.emit()`

- `.emit("event", data, ...)` triggers a new event
- The first argument is the type of event, same as `.on`
- Any subsequent arguments become arguments in `.on`'s callback function(s)
- One event emit could trigger an infinite number of listeners asynchronously

```
// The "some-event" handler from before will get called
emitter.emit("some-event", "hello world!");
```

Unlistening with `.removeListener()`

- Takes two args, `.removeListener("type", callbackFn)`
- Will stop a listener from calling a specific callback
- You **must** provide the same event type & a reference to the same callback function if you want to remove the listener
- Therefore If you want to remove a listener, you must name the function so that it can be removed

Unlistening with `.removeListener()` (Example)

```
function printMessage(message) {
  console.log(message);
}

emitter.on("some-event", printMessage);

// Later on...

emitter.removeListener("some-event", printMessage);
```

Unbinding all listeners with `removeAllListeners()`

- Only takes in one argument, the event type
- Should rarely be used, might have unintended effects

```
// After removeAllListeners, nothing will
// react to "some-event" emits
emitter.removeAllListeners("some-event");
emitter.emit("some-event", "Nothing will happen");
```

Extending `EventEmitter`s via inheritance

- `EventEmitter` is a class, just like our Songs from Jukebox
- Which means we can `extends` it, giving us its functionality

```
class Doorbell extends EventEmitter {
  ring() {
    this.emit("ring");
  }
}

var db = new Doorbell();

db.on("ring", function() {
  console.log("Ding dong!");
});

db.ring();
```

Caveats

- Callbacks will only work if bound *before* an event emits
  - If you listen before an event is emitted, nothing happens
- Event listener callbacks are called *asynchronously*, so you can't rely on the order the callbacks get called
- Event listeners can be bound twice with the same function
  - Unlistening will only unbind one of them each time

Let's create an `Elevator` class together!

- Make a new file, `elevator.js`, that will export our `Elevator` class as a module
- It should extend `EventEmitter` so that we can emit & listen to events
- This is a small and simple elevator, so it can only hold one `currentPassenger` at a time by passing them to `loadPassenger(Passenger)`
  - `Passenger` is a class that we'll make in `passenger.js`
  - Each `Passenger` will have a `name` and a `desiredFloor`
  - We should also be able to call `unloadPassenger()` to remove them from the elevator
- The Elevator should be able to `goUp()` and `goDown()`
  - This should mark the elevator as `isMoving`
  - After 1 second, we should change the elevator's `currentFloor` by +1 or -1
  - Also after 1s, we'll emit `"up"` / `"down"` events that provide the current passenger and floor

Now it's your turn: Make it work!

- Import the `Elevator` and `Passenger` classes
- Create 3 `Passengers` with different names and desired floors, stored in an array
- For each `Passenger` instance, we're going to:
  - Load them in to the elevator by `pop()`ping them off the array
  - `goUp()` until we're at their `desiredFloor`
  - `unloadPassenger()` to let them off
  - `goDown()` until we're back at the lobby (`currentFloor === 0`)
  - `loadPassenger()` the next person (also by popping them off)
- Repeat this until all of the passengers are unloaded and we're back at the lobby.
- Every action should be `console.log()`'d

**Note, you cannot do this synchronously with a for loop. You must use the event emitters.*