1 Build a Back-End with Node/Express.js

1.1 Introduction

1.2 Node REPL

- Is an abbrebivation for Read-eval-print loop
- Node comes with built-in javascript REPL
- `.editor` goes into editor mode
  - Use `CTRL + D` when ready to evaluate the input
- A REPL can be extremely useful for performing calculations
- The Node environment contains a number of Node-specific `global` elements in addition to those built into the JavaScript language
  - can be examined using command `console.log(global)`

1.3 Running a Program with Node

- Done using command `node myProgram.js`
- Javascript code is written to file `.js` extension

1.4 Accessing the Process Object

- Node has a global `process` object with useful methods and information about the current process.
  - `process.env` property is an object which stores and controls information about the environment in which the process is currently running
    - `PWD` - holds a string with the directory where the current process is located
    - `NODE_ENV` - holds a value of either production or development

    Example

```
1    if (process.env.NODE_ENV === 'development'){
2        console.log('Testing! Testing! Does everything work?');
3    }
4
```

    - `process.memoryUsage()` returns information on the CPU demands of the current process.
    - `process.memoryUsage().heapUsed` return a number representing how many bytes of memory the current process is using.

- `process.argv` property holds an array of command line values provided when the current process was initiated
  - first element in the array is the absolute path to Node
  - second element in the array is the path to the file that's running
  - following elements will be any command line arguments provided when the process was initiated (like C)!!!

```
node myProgram.js testing several features

```

```
console.log(process.argv[3]); // Prints 'several'

```

1.5 Core Modules and Local Modules

- **Modularity** is a software design technique where one program has distinct parts each providing a single piece of the overall functionality.
  - Is essential when creating scalable programs
    - incorporate libraries and frameworks and separate the program's concerns into manageable chunks
- **Modules** come together to build a cohesive whole
  - is imported using `require()`

```
// Require in the 'events' core module:
let events = require('events');

```

  - is exported using `module.exports`

```
module.exports = class Dog {

    constructor(name) {
        this.name = name;
    }

    praise() {
        return `Good dog, ${this.name}!`;
    }
};

```

1.6 Node Package Manager

- NPM, which stands for Node Package Manager

1.7 Event-Driven Architecture

- Node is often described as having event-driven architecture
- Node provides an `EventEmitter` class which we can access by requiring in the events core module

```
// Require in the 'events' core module
let events = require('events');

// Create an instance of the EventEmitter class
let myEmitter = new events.EventEmitter();

```

1.8 Event-Driven Architecture

- Node is often described as having event-driven architecture.
  - This feels so much like threaded programming
- event emitter instance has an `.on()` method which assigns a listener callback function to a named event.
  - first argument - the name of the event as a string
  - second argument - the listener callback function

```

let newUserListener = (data) => {
    console.log(`We have a new user: ${data}.`);
};

// Assign the newUserListener function as the listener callback for 'new user' events
myEmitter.on('new user', newUserListener)

// Emit a 'new user' event
myEmitter.emit('new user', 'Lily Pad') //newUserListener will be invoked with 'Lily Pad'

```

1.9 Asynchronous JavaScript with Node.js

- Node was designed to use an event loop like the one used in browser-based JavaScript execution
- The event-loop enables asynchronous actions to be handled in a non-blocking way.
  - APIs trigger the subscription to and emitting of events to signal the completion of the operation

Example

```
    let keepGoing = true;

    let callback = () => {
        keepGoing = false;
    };

    setTimeout(callback, 1000); // Run callback after 1000ms

    while(keepGoing === true) {
        console.log('This is the song that never ends. Yes, it just goes on and on my friends. Some people started singing it, not knowing what it was, and they'll continue singing it forever just because...')
    };

```

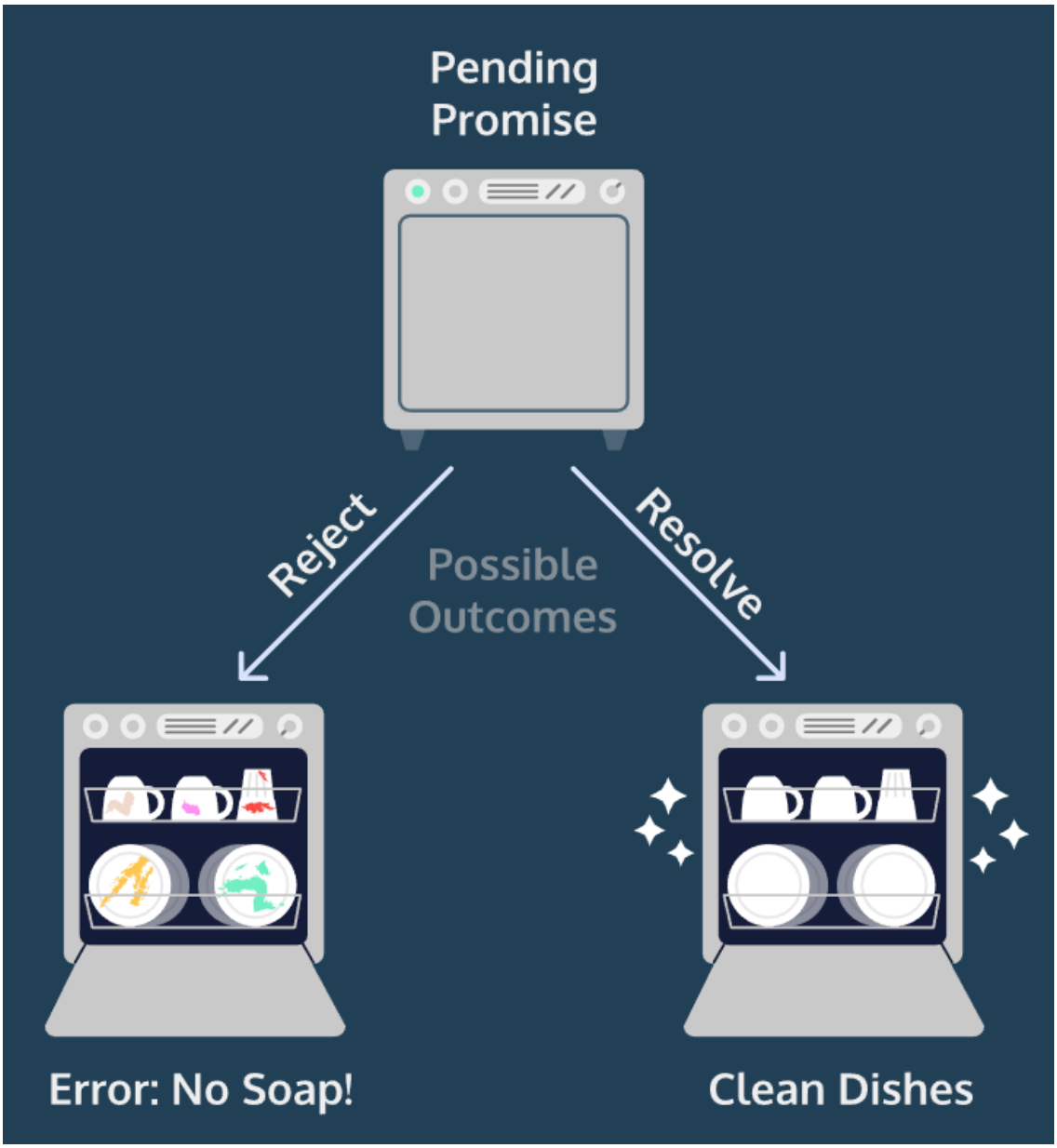
  - The while loop will continue forever
  - Why? because no signal has been sent
  - To resolve this issue, replace `setTimeout` with `setTimeInterval`

- `Promise`, `async ... await`
  - modern way of handling asynchronous tasks

1.10 Asynchronous Javascript - Introduction

- **asynchronous operation** is one that allows the computer to "move on" to other tasks while waiting for the asynchronous operation to complete.

1.11 Asynchronous Javascript - What is a Promise?



- **Pending:** The initial state— the operation has not completed yet.
- **Fulfilled:** The initial state— the operation has not completed yet.
- **Rejected:** The initial state— the operation has not completed yet.

1.12 Constructing a Promise Object

- use the `new` keyword and the `Promise` constructor method

```
1    const executorFunction = (resolve, reject) => { };
2    const myFirstPromise = new Promise(executorFunction);
3
```

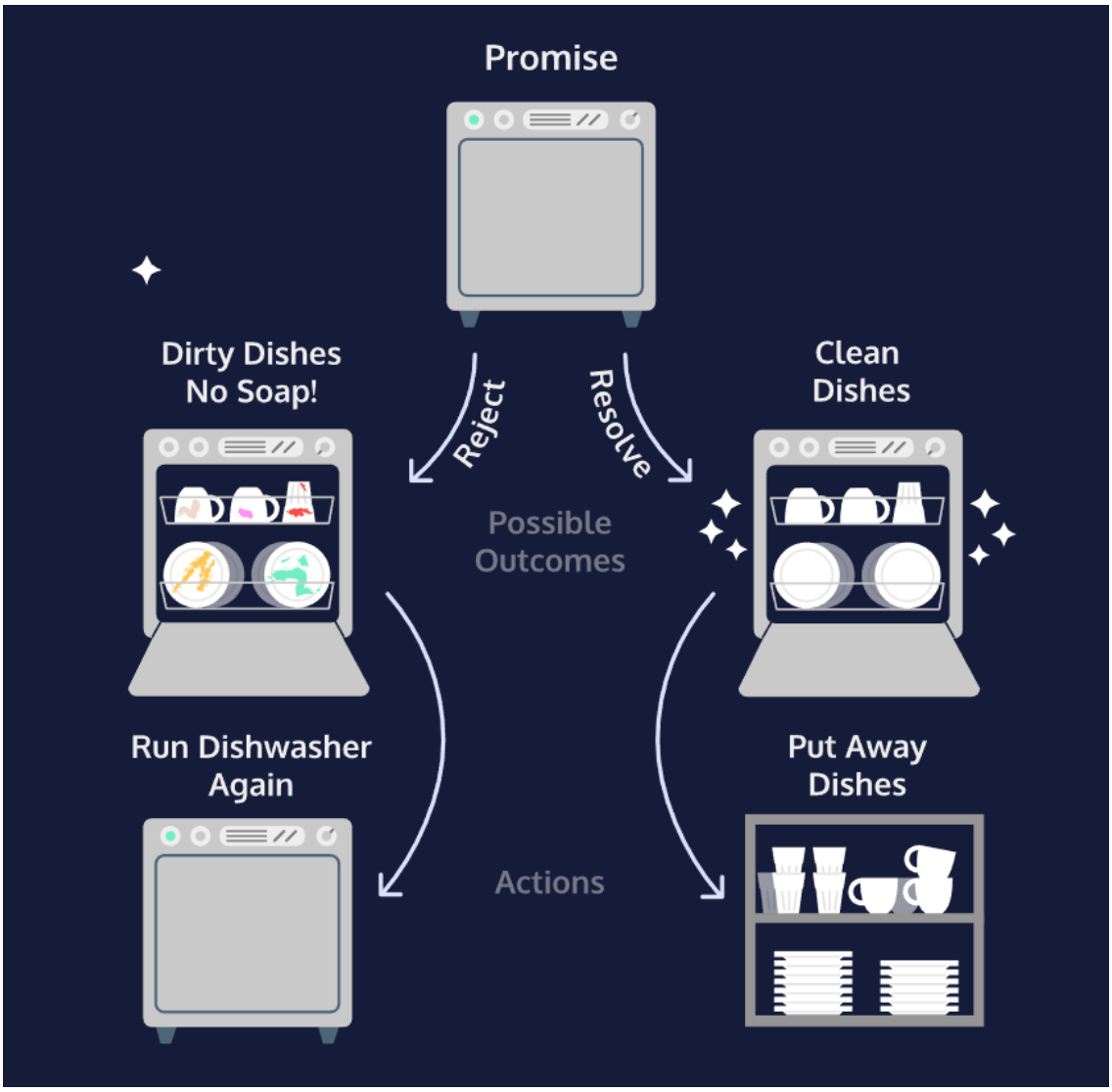
- `Promise` constructor method takes a function parameter called the `executor` function which runs automatically when the constructor is called
- The `executor` function has has two function parameters
  - `resolve()`
    - is a function with one arguement
    - invoke causes the promise's status to change from `pending` to `fulfilled`
    - sets promises' resolved value to be the arguement passed to `resolve`
  - `reject()`
    - takes a reason or error as an argument
    - invoke causes `reject()` to change the promise's status from `pending` to `rejected`

Example

```
const executorFunction = (resolve, reject) => {
    if (someCondition) {
        resolve('I resolved!');
    } else {
        reject('I rejected!');
    }
}
const myFirstPromise = new Promise(executorFunction);

```

1.13 The Node setTimeout() Function



- Rather than constructing promises, you'll be handling `Promise` objects returned to you as the result of an asynchronous operation
  - It will start off pending but settle eventually.
- `setTimeout()`
  - Is a node API
  - Has two parameters
    - a callback function
    - a delay in milliseconds.

```
        const executorFunction = (resolve, reject) => {
            if (someCondition) {
                resolve('I resolved!');
```

```
            } else {
                reject('I rejected!');
            }
        }
        const myFirstPromise = new Promise(executorFunction);

```

- the embedded code will exectue after said time (not exactly at the time)
  - because the function within is added to a line of code waiting to be run

1.14 Consuming Promises

- `.then()` is that it always returns a promise. We'll return to this in more detail in a later exercise and explore why it's so important.
- `.then()` takes two callback functions as arguements
  - First argument - is the success handler
  - Second argument - is the failure handler

1.15 The onFulfilled and onRejected Functions

1.16 Using catch() with Promises

- `separation of concerns` - one way to write cleaner code
- Javascript doesn't mind whitespace
- `.catch()` function takes only one argument, onRejected

Example

```
prom
.then((resolvedValue) => {
    console.log(resolvedValue);
})
.catch((rejectionReason) => {
    console.log(rejectionReason);
});

```

1.17 Chaining Multiple Promises

- `composition` - the process of chaining promises together
- Promise is designed with composition in mind

Example

```
    firstPromiseFunction()
    .then((firstResolveVal) => {
        return secondPromiseFunction(firstResolveVal);
    })
    .then((secondResolveVal) => {
        console.log(secondResolveVal);
    });

```

- We invoke a function `firstPromiseFunction()` which returns a promise.
- `.then()` is invoked with an anonymous function as the success handler
- Inside the success handler we return a new promise— the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- second `.then()` is invoked to handle the logic for the second promise settling.
- Inside second `.then()`, we have a success handler which will log the second promise's resolved value to the console.

```
const {checkInventory, processPayment, shipOrder} = require('./library.js');

const order = {
    items: [['sunglasses', 1], ['bags', 2]],
    giftcardBalance: 79.82
};

checkInventory(order)
.then((resolvedValueArray) => {
    // Write the correct return statement here:
    return processPayment(resolvedValueArray);
})
.then((resolvedValueArray) => {
    // Write the correct return statement here:
    return shipOrder(resolvedValueArray);
})
.then((successMessage) => {
    console.log(successMessage);
})
.catch((errorMessage) => {
    console.log(errorMessage);
});


```

1.18 Avoiding Common Mistakes

- **Mistake 1:** Nesting promises instead of chaining them.
  - Works fine

  - Imagine if we are handling five or then promises

```
    returnsFirstPromise()
    .then((firstResolveVal) => {
        return returnsSecondValue(firstResolveVal)
        .then((secondResolveVal) => {
            console.log(secondResolveVal);
        })
    })

```

- **Mistake 2:** Forgetting to return a promise.

  -

```
    returnsFirstPromise()
    .then((firstResolveVal) => {
        returnsSecondValue(firstResolveVal)
    })
    .then((someVal) => {
        console.log(someVal);
    })

```

  - `returnsFirstPromise()` which returns a promise.
  - invoke a second `.then()`. Since we didn't return, this `.then()` is invoked on a promise with the same settled value as the original promise

```
 const {checkInventory, processPayment, shipOrder} = require('./library.js
');

 const order = {
     items: [['sunglasses', 1], ['bags', 2]],
     giftcardBalance: 79.82
 };

 // Refactor the code below:

 checkInventory(order)
     .then((resolvedValueArray) => {
         return processPayment(resolvedValueArray);
     })
     .then((resolvedValueArray) => {
         return shipOrder(resolvedValueArray);
     })
     .then((successMessage) => {
         console.log(successMessage);
     });

```

1.19 Using Promise.all()

- promise composition is a great way to handle situations where asynchronous operations depend on each other or execution order matters
- What if we don't care about order? - make simple using `Promise.all()`
- `Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:
  - If every promise in the argument array resolves, the single promise returned from Promise.all() will resolve with an array containing the resolve value from each promise in the argument array.
  - If any promise from the argument array rejects, the single promise returned from Promise.all() will immediately reject with the reason that promise rejected.
    - This behavior is sometimes referred to as **failing fast.**

Example

```
let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(), returnsPromThree()]);

myPromises
    .then((arrayOfValues) => {
    console.log(arrayOfValues);
    })
    .catch((rejectionReason) => {
    console.log(rejectionReason);
    });

```

  - `myPromises` assigned to invoking `Promise.all()`.
  - `Promise.all()` has an array of three promises— the returned values from functions.
  - `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
  - `.catch()` with a failure handler which will print the first rejection message if any promise rejects.

```
    const {checkAvailability} = require('./library.js');

    const onFulfill = (itemsArray) => {
      console.log(`Items checked: ${itemsArray}`);
      console.log(`Every item was available from the distributor. Placing order now.`);
    };

    const onReject = (rejectionReason) => {
        console.log(rejectionReason);
```

```
10        };
11
12        // Write your code below:
13
14        const checkSunglasses = checkAvailability('sunglasses', 'Favorite
     Supply Co.');
15        const checkPants = checkAvailability('pants', 'Favorite Supply Co
    .');
16        const  checkBags = checkAvailability('bags', 'Favorite Supply Co
    .');
17
18        Promise.all([checkSunglasses, checkPants, checkBags])
19          .then(onFulfill)
20          .catch(onReject);
21
```

1.20 Review

- Promises can be in one of three states: pending, resolved, or rejected.
- `.then()` is a success handler callback containing the logic for what should happen if a promise resolves
- `.catch()` is a failure handler callback containing the logic for what should happen if a promise rejects.
- `setTimeout()` is a Node function which delays the execution of a callback function using the event-loop
- To take advantage of concurrency, we can use `Promise.all()`

1.21 Quiz

1. How many parameters does a Promise constructor take?

   ```
   const example = new Promise( ? ? ? );
   ```

   Answer: 1

2. Which of the executor function's parameter is called if the asynchronous task completes successfully?

   ```
   const example = new Promise( (function1, function2) =>  . . .  );
   ```

   Answer: function1

3. What is the fulfilled value of `Promise.all()`?

   Answer: An array

4. What is the fulfilled value of `Promise.all()`?

   Answer: An array

5. What value is printed to the console?

```
const asyncHello = new Promise((resolve, reject) => {
    setTimeout(resolve, 1000, 'Hello!');
});

console.log(typeof asyncHello);

```

   Answer: An object

6. True or False: The `.then()` method returns a Promise.

   Answer: True

7. What is value of the argument that is passed to the onReject()?

```
let onFulfill = value => {console.log(value)};
let onReject = reason => {console.log(reason)};

const promise =  new Promise( (resolve, reject) => {
    if (false) {
    resolve('success value');
    } else {
    reject();
    }
});

promise.then(onFulfill, onReject);

```

   Answer: undefined

8. True or False: promise1 and promise2 both produce the same output.

```
 const examplePromise1 = new Promise((resolve, reject) => { reject('Uh
-oh!') });
 const examplePromise2 = new Promise((resolve, reject) => { reject('Uh
-oh!') });

 const onFulfill = value => {console.log(value)};
 const onReject = reason => {console.log(reason)};

 const promise1 = examplePromise1.then(onFulfill, onReject);

 const promise2 = examplePromise2.then(onFulfill).catch(onReject);

```

Answer: True

9. True or False: promise1 and promise2 both produce the same output.

```
 const examplePromise1 = new Promise((resolve, reject) => { reject('Uh-oh!') });
 const examplePromise2 = new Promise((resolve, reject) => { reject('Uh-oh!') });

 const onFulfill = value => {console.log(value)};
 const onReject = reason => {console.log(reason)};

 const promise1 = examplePromise1.then(onFulfill, onReject);

 const promise2 = examplePromise2.then(onFulfill).catch(onReject);

```

Answer: True

10. Which one of the following is NOT a state that a Promise resolves to?

Answer: Undefined

11. What state will this promise be in after 0 seconds?

```
const examplePromise = () => {
    return new Promise((resolve, reject) => {
        if (true) {
        setTimeout( () => resolve('success'), 3000);
        } else {
        setTimeout( () => resolve('failed'), 5000);
        }
    });
};

```

Answer: Pending

12. What will be printed to the console after running the code provided?

```
let link = state => {
    return new Promise(function(resolve, reject) {
        if (state) {
        resolve('success');
        } else {
        reject('error');
        }
    });
    }

    let promiseChain = link(true);

```

```
        promiseChain
        .then( data => {
            console.log(data + " 1");
            return link(true);
        })
        .then( data => {
            console.log(data+ " 2");
            return link(true);
        });

```

Answer:

success1
success2

2 Asynchronous Javascript - ASYNC AWAIT

2.1 Introduction

- JavaScript is non-blocking
  - instead of stopping the execution of code while it waits, JavaScript uses an event-loop which allows it to efficiently execute other tasks while it awaits the completion of these asynchronous actions.
- `async ... await` is a syntatic sugar

2.2 The Async Keyword

- `async` keyword is used to write functions that handle asynchronous actions
- Syntax
  - Original

```
    async function myFunc() {
        // Function body here
    };

    myFunc();

```

  - Function Expression

```
    const myFunc = async () => {
        // Function body here
    };

    myFunc();

```

- `async` function always returns a promise
  - We use traditional promise syntax
    * `.then()`
    * `.catch`
    * `async`
- Returns in one of three ways
  - `undefined` - If there's nothing returned from the function
  - If non-promise value is returned from function, it will return a promise resolved to the value
  - If promise is returned from the function, it will simply return that value

```
function withConstructor(num){
    return new Promise((resolve, reject) => {
      if (num === 0){
        resolve('zero');
      } else {
        resolve('not zero');
      }
    })
  }

withConstructor(0)
    .then((resolveValue) => {
    console.log(` withConstructor(0) returned a promise which resolved to: ${resolveValue}.`);
})

// Write your code below:
async function withAsync(num) {
        return new Promise((resolve, reject) => {
        if (num === 0){
            resolve('zero');
        } else {
            resolve('not zero');
        }
    })
};


// Leave this commented out until step 3:
/*
withAsync(100)
    .then((resolveValue) => {
    console.log(` withAsync(100) returned a promise which resolved to: ${resolveValue}.`);
})
*/
```

2.3 The await Operator

- `async` keyword, by itself, it doesn't do much
- `await` keyword, is used inside an `async` function
- `await` halts, or pauses, the execution of our `async` function until a given promise is resolved

Example

```
    async function asyncFuncExample(){
        let resolvedValue = await myPromise();
        console.log(resolvedValue);
    }

    asyncFuncExample(); // Prints: I am resolved now!

```

```
const brainstormDinner = require('./library.js')


// Native promise version:
function nativePromiseDinner() {
  brainstormDinner().then((meal) => {
      console.log(`I'm going to make ${meal} for dinner.`);
  })
}


// async/await version:
async function announceDinner() {
  // Write your code below:
  let meal = await brainstormDinner();
  console.log(`I'm going to make ${meal} for dinner.`);
}

announceDinner();
```

2.4 Writing async Functions

- `await` keyword halts the execution of an async function until a promise is no longer pending

```
    let myPromise = () => {
        return new Promise((resolve, reject) => {
            setTimeout(() => {
            resolve('Yay, I resolved!')
            }, 1000);
        });
    }

```

```
    async function noAwait() {
        let value = myPromise();
       console.log(value);
    }

    async function yesAwait() {
        let value = await myPromise();
        console.log(value);
    }

    noAwait(); // Prints: Promise { <pending> }
    yesAwait(); // Prints: Yay, I resolved!

```

- `noAwait()` function logs Promise ¡pending¿ to the console
- `yesAwait()` returns resolved value of promise
- IMPORTANT!! `await` operator returns the resolved value of a promise

```
const shopForBeans = require('./library.js');

async function getBeans() {
  console.log(`1. Heading to the store to buy beans...`);
  let value = await shopForBeans();
  console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);
}

getBeans();
```

2.5 Handling Dependent Promises

- The true beauty of async...await is when we have a series of asynchronous actions which depend on one another
- With native promise syntax, we use a chain of `.then()` functions making sure to return correctly each one:
  - Code is a lot simpler

```
        function nativePromiseVersion() {
            return FirstPromise()
            .then((firstValue) => {
                console.log(firstValue);
                return returnsSecondPromise(firstValue);
            })
           .then((secondValue) => {
                console.log(secondValue);
            });
        }

```

```
        async function asyncAwaitVersion() {
            let firstValue = await returnsFirstPromise();
            console.log(firstValue);
            let secondValue = await returnsSecondPromise(firstValue);
            console.log(secondValue);
        }

```

```
 const {shopForBeans, soakTheBeans, cookTheBeans} = require('./library.js')
;

 // Write your code below:
 async function makeBeans() {
   let type = await shopForBeans();
   let isSoft = await soakTheBeans(type);
   let dinner = await cookTheBeans(isSoft);

   console.log(dinner);
 }

 makeBeans();
```

2.6 Handling Errors

- `.catch()` is used with a long promise chain - there is no indication of where in the chain the error was thrown
- `async ... await` it's easier to debug

  Example

```
        async function usingTryCatch() {
            try {
              let resolveValue = await asyncFunction('thing that will
   fail');
              let secondValue = await secondAsyncFunction(resolveValue);
            } catch (err) {
              // Catches any errors in the try block
              console.log(err);
            }
        }

        usingTryCatch();

```

  Example 2 (same but using traditional catch)

```
    async function usingPromiseCatch() {
        let resolveValue = await asyncFunction('thing that will fail');
        }

        let rejectedPromise = usingPromiseCatch();
        rejectedPromise.catch((rejectValue) => {
```

```
        console.log(rejectValue);
        })

```

•

```
const cookBeanSouffle = require('./library.js');

// Write your code below:

async function hostDinnerParty() {
  try {
    let dinner = await cookBeanSouffle();
    console.log(`${dinner} is served!`);
  } catch (error) {
    console.log(error);
    console.log('Ordering a pizza!');
  }
}

hostDinnerParty();
```

2.7 Handling Independent Promises

- `async await` takes advantage of concurrency
- Here the code works synchronously. Wait for `firstAsyncThing` and then `secondAsyncThing`

```
        async function waiting() {
        const firstValue = await firstAsyncThing();
        const secondValue = await secondAsyncThing();
        console.log(firstValue, secondValue);
        }

```

- Here, `firstAsyncThing` and `secondAsyncThing` are called asynchronously and waits in `console.log` until all are ready

```
        async function concurrent() {
        const firstPromise = firstAsyncThing();
        const secondPromise = secondAsyncThing();
        console.log(await firstPromise, await secondPromise);
        }

```

- to execute fully in parallel, we must use individual `.then()` functions and avoid halting our execution with await.

```
        async function concurrent() {
        const firstPromise = firstAsyncThing();
        firstPromise.then((resolved) => {
            ...
```

```
        });
        const secondPromise = secondAsyncThing();
        console.log(await firstPromise, await secondPromise);
        }

```

```
 let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library
.js')

 // Write your code below:
 async function serveDinner() {
   let vegetablePromise = steamBroccoli();
   let starchPromise = cookRice();
   let proteinPromise = bakeChicken();
   let sidePromise = cookBeans();

   console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await sidePromise}.`);
 }

 serveDinner();
```

2.8 Await Promise.all()

- executes multiple promises simultaneously
- returns a single promise
- promise's resolve value will be an array containing the resolved values of each promise from the argument array
- also has the benefit of **failing fast**
  - meaning it won't wait for the rest of the asynchronous actions to complete once any one has rejected.

```
        async function asyncPromAll() {
            const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);
            for (let i = 0; i<resultArray.length; i++){
              console.log(resultArray[i]);
            }
        }

```

```
 let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library
.js')

 // Write your code below:
 async function serveDinnerAgain() {
```

```
   const foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

   let vegetable = foodArray[0];
   let starch =  foodArray[1];
   let protein =  foodArray[2];
   let side =  foodArray[3];

   console.log(`Dinner is served. We're having ${vegetable}, ${starch}, ${protein}, and ${side}.`);
 }
```

2.9 Quiz

1. Which of the following is useful for awaiting multiple promises where all are required but none depend on each other to execute?

   Answer: `Promise.all()`

2. What purpose does the async keyword serve?

   Answer: It's a keyword which indicates an asynchronous function.

3. What purpose does the await keyword serve?

   Answer: It's an operator used only inside an async function that halts the execution of a function until a given promise is no longer pending and returns the resolved value of the promise.

4. Which of the following is NOT a benefit of the async...await syntax?

   Answer: It causes promises to resolve faster.

5. Which of the following is NOT a benefit of the async...await syntax?

```
async function myFunction() {
    return 'hello world';
  }

  myFunction()
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
```

   Answer: hello world

6. Which of the following is NOT a benefit of the async...await syntax?

```
async function example() { }
```

Answer: undefined

Correct Answer: Promise

7. Given that firstPromise(), secondPromise(), and thirdPromise() do not depend on each other to execute, what isn't ideal about the provided code?

```
async function threePromises() {
    let first = await firstPromise();
    let second = await secondPromise();
    let third = await thirdPromise();
    console.log(first, second, third);
  }
```

Answer: Using await halts the execution of the function which means consecutive promises would wait to execute until the previous promise resolved. Instead, we should allow for concurrency whenever possible.

8. True or False: the async...await syntax has functionality that cannot be accomplished by native promises.

Answer: True

Correct Answer: False (async ... await is a syntatic sugar)

2.10 User Input/Output

- In node.js, `console.log()` is a thin wrapper of `.stdout.write()` method of the `process object`
- `stdin.on()` - receive input from a user through the terminal

```
process.stdin.on('data', (userInput) => {
    let input = userInput.toString()
    console.log(input)
  });

```

  - `.on()` can be used
    * `process.stdin` is an instance of `EventEmitter`
    *
    * 'data' event will be fired on when a user enters text into the terminal and hits enter

```
let {testNumber} = require('./game.js');

process.stdout.write("I'm thinking of a number from 1 through 10. What do
you think it is? \n(Write \"quit\" to give up.)\n\nIs the number ... ");

let playGame = (userInput) => {
  let input  = userInput.toString().trim();
    testNumber(input);
};

process.stdin.on('data', playGame);
```

2.11 Errors

- node has all javascript errors
  - EvalError
  - SyntaxError
  - RangeError
  - ReferenceError
  - TypeError
  - URIError
  - Error
- `try...catch` statements can be used for error handling (but doesn't work with node's async behavior)
- errors can be generated and be `thrown`

```
        const errorFirstCallback = (err, data)  => {
            if (err) {
              console.log(`There WAS an error: ${err}`);
            } else {
               // err was falsy
                console.log(`There was NO error. Event data: ${data}`);
            }
        }

```

- If the asynchronous task results in an error, it will be passed in as the first argument to the callback function. If no error was thrown, the first argument will be `undefined`.

```
const api = require('./api.js');

// An error-first callback
let errorFirstCallback = (err, data) => {
  if (err) {
    console.log(`Something went wrong. ${err}\n`);
```

```
    } else {
      console.log(`Something went right. Data: ${data}\n`);
    }
  };

  api.errorProneAsyncApi('problematic input', errorFirstCallback);
```

2.12 Filesystem

- `sandboxing` - allows to have only limited access to a user's filesystem
  - Is important in javascript
  - Not important in node.js

```
  const fs = require('fs');

  let readDataCallback = (err, data) => {
    if (err) {
      console.log(`Something went wrong: ${err}`);
    } else {
      console.log(`Provided file contained: ${data}`);
    }
  };

  fs.readFile('./file.txt', 'utf-8', readDataCallback);
```

```
const fs = require('fs');

let secretWord = 'cheeseburgerpizzabagels';
let readDataCallback = (err, data) => {
  if (err) {
    console.log(`Something went wrong: ${err}`);
  } else {
    console.log(`Provided file contained: ${data}`);
  }
};

 fs.readFile('./fileOne.txt', 'utf-8', readDataCallback);
 fs.readFile('./anotherFile.txt', 'utf-8', readDataCallback);
 fs.readFile('./finalFile.txt', 'utf-8', readDataCallback);
```

2.13 Readable Streams

- data is sequentially, piece by piece, in what is known as a **stream**
  - preferable since you don't need enough RAM to process all the data
- To read files line-by-line, we can use the `.createInterface()` method from the `readline` core module

  - `.createInterface()` returns an `EventEmitter` set up to emit 'line' events
  - `fs.createReadStream(...)` creates a stream from the ... file.

```
const readline = require('readline');
const fs = require('fs');

const myInterface = readline.createInterface({
  input: fs.createReadStream('text.txt')
});

myInterface.on('line', (fileLine) => {
  console.log(`The line read: ${fileLine}`);
});
```

```
const readline = require('readline');
const fs = require('fs');

const printData = (data) => {
  console.log(`Item: ${data}`);
}

const myInterface = readline.createInterface({
  input: fs.createReadStream('shoppingList.txt')
});

 myInterface.on('line', printData);
```

2.14 Writable Streams

- `fs.createWriteStream()` - create a writeable stream to a file

```
const fs = require('fs')

const fileStream = fs.createWriteStream('output.txt');

fileStream.write('This is the first line!');
fileStream.write('This is the second line!');
fileStream.end();
```

- `.end()` indicates the end of a writable stream