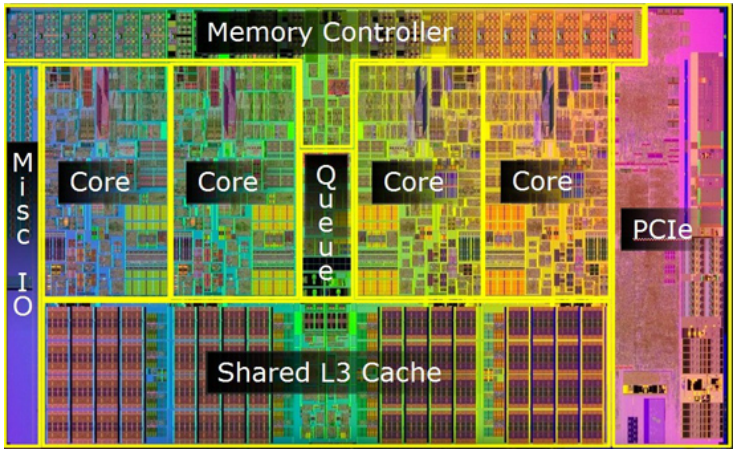
JavaScript Part 2
Async

Samuel Cho, Ph.D.

NKU ASE/CS

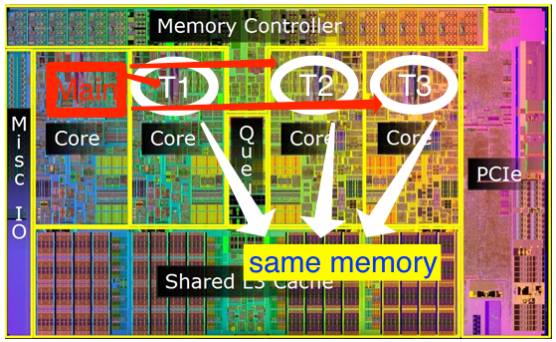
Thread Model

Multi-Core CPU



- In our computer, we have a CPU that has multiple cores.
- It means we need to make an application that can be run on these cores.

Multi-Thread Programming



- In multi-thread programming, a main (Main) process spawns multiple threads (T1, T2, T3) that share memory.
- Each thread can be executed on any of the cores.

Java - Multi-Thread Language

- Java is a multi-thread programming language.
- Using the Thread class, we can implement the main code in the run() method.

Code #

```
1 class Multi extends Thread {
2   public void run()
3   {
4     try { ... }
5     catch (Exception e) { ... }
6   }
7 }
```

- We can spawn as many threads as we want in the main method.
- It is the OS's job to execute each thread; it decides on what core a thread is run.

Code #

```
1 public class Multithread {
2   public static void main(String[] args) {
3     int n = 8; // Number of threads
4     for (int i = 0; i < n; i++) {
5       Multi object = new Multi();
6       object.start();
7     }
8   }
9 }
```

- Most languages support this multi-thread programming model.
- These are examples from C#, but other programming languages are almost identical.

Code #

```
1  using System; using System.Threading;
2  namespace Threads
3  {
4    class Program {
5      public static void Main(string[] args) {
6        Thread thread = new Thread(new ThreadStart(Task));
7        thread.Start(); ...
8        thread.Join();
9      }
10     private static void Task() { ... }
11 }
```

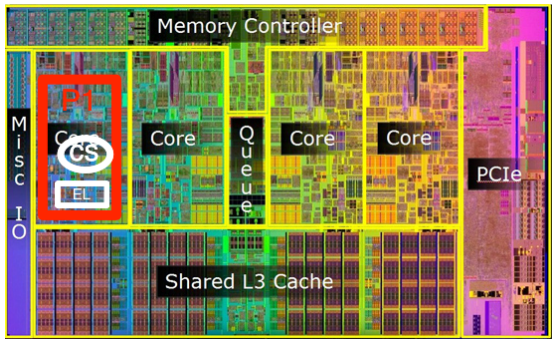
The Issue of This Model

However, this approach has a big issue: It is (very) hard to make multi-thread programs work correctly.

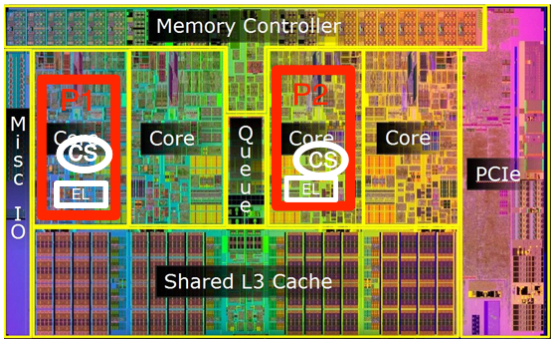
- We have to manage threads.
- We have to manage the memory access by each thread, and if this is wrong, the program does not work.
- It is extremely hard to debug multi-core programming.

JavaScript Single Thread Model

The Solution - Single Thread Model



- We can address this issue by "not" using multiple threads, but using a single thread.
- In this model, a process has one thread, but it manages call stack (CS) and event loop (EL) to support asynchronous I/O.



- On the multiple core, we can spawn a new process (not a thread) and the process manages its single thread with the call stack and event loop.

JavaScript: Single Thread Model

- JavaScript uses 'Single Thread Model.'
- However, in JavaScript, we can implement an asynchronous program that executes multiple functions asynchronously.

- In this example, the setTimeout() function executes the lambda expression in 1000 ms, but line 5 is executed without waiting for its results.

Code #

```
1 console.log("first")
2 setTimeout(() => {
3     console.log("
        ↪ second")
4 }, 1000)
5 console.log("third")
```

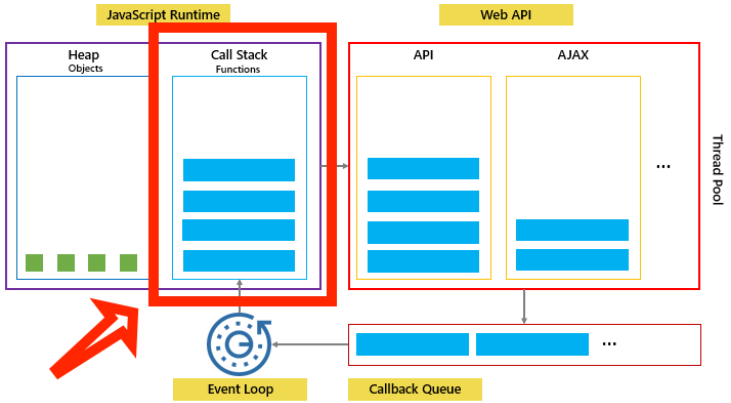
Command > Results

```
first
third
second
```

JavaScript Call Stack

- JavaScript single-thread model has a call stack and event loop.
- The JavaScript engine executes the JavaScript code from the top of the file and works its way down.

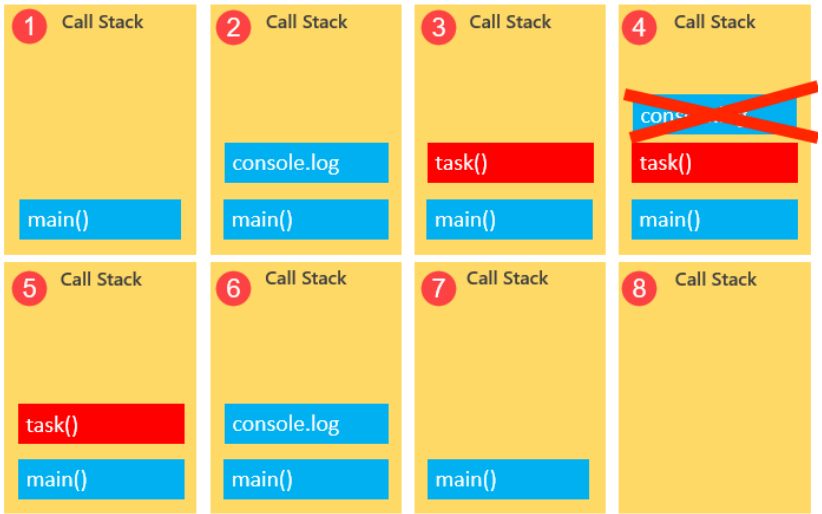
- It creates the execution contexts, pushes, and pops functions onto and off the call stack when it executes code.



- In this example, console.log() function is pushed in the call stack and popped out (②) in line 7.

Code #

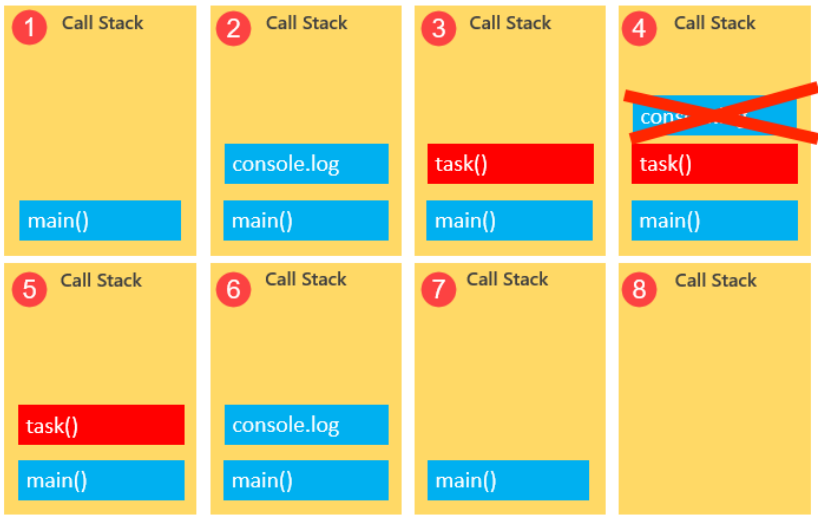
```
1  function task(message) {
2  // waiting for input from users
3       ...
4       console.log(message);
5  }
6
7  console.log('Start script...');
8  task('Call an API');
9  console.log('Done!');
```



- For the function task, it blocks other functions to be executed (④) until users give inputs in line 8, as the console.log() function can be executed only after the task() function (⑥) in line 9.

Code #

```
1  function task(message) {
2  // waiting for input from users
3       ...
4       console.log(message);
5  }
6
7  console.log('Start script...');
8  task('Call an API');
9  console.log('Done!');
```

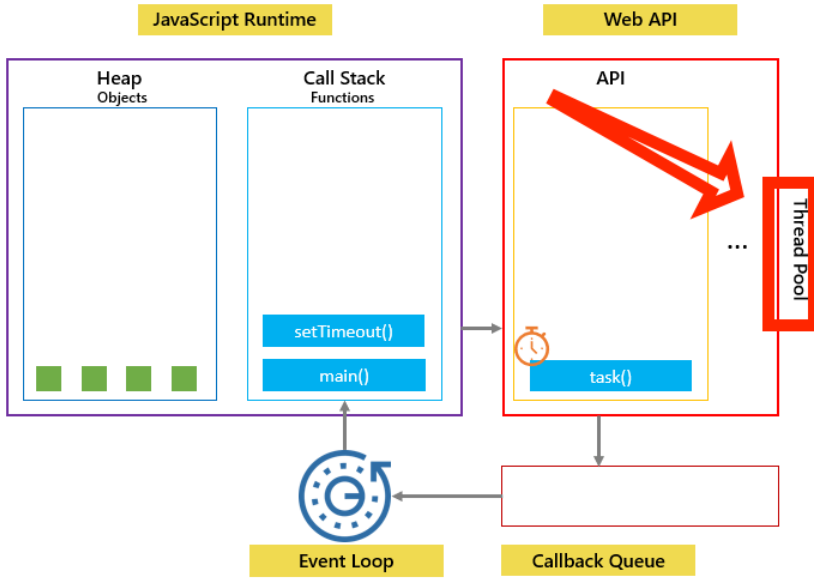


- This is an issue, as the core is idling when it waits for users' input.
- In this case, we can use the callback queue.

- In this example, when calling the setTimeout() function, the JavaScript engine places the lambda expression on the Thread pool stack, and the Web API creates a timer that expires in 1 second in the Thread Pool.

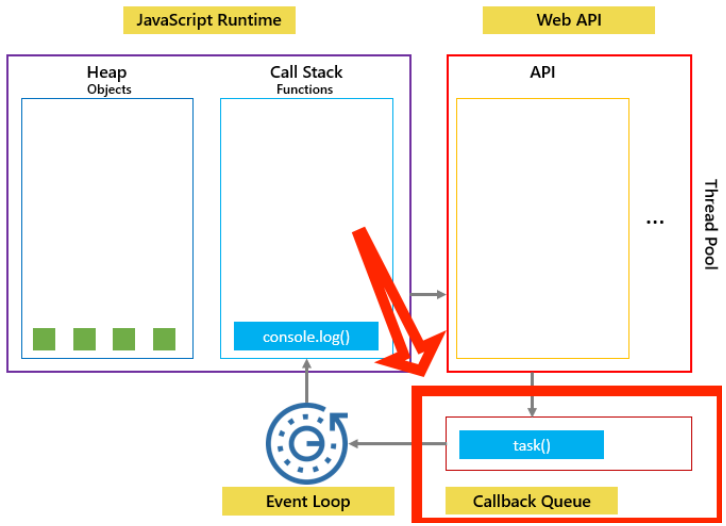
Code #

```
1 console.log('Start script...');
2 setTimeout(() => {
3     task('Download a file.');
4 }, 1000);
5 console.log('Done!');
```

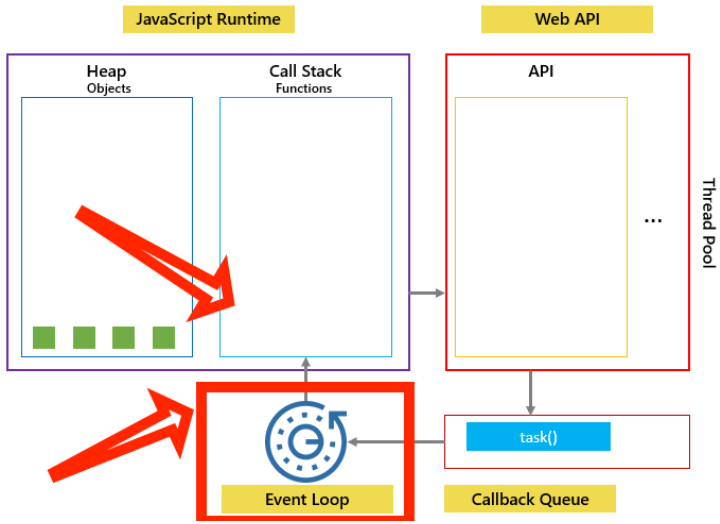


Callback Queue and Event Loop

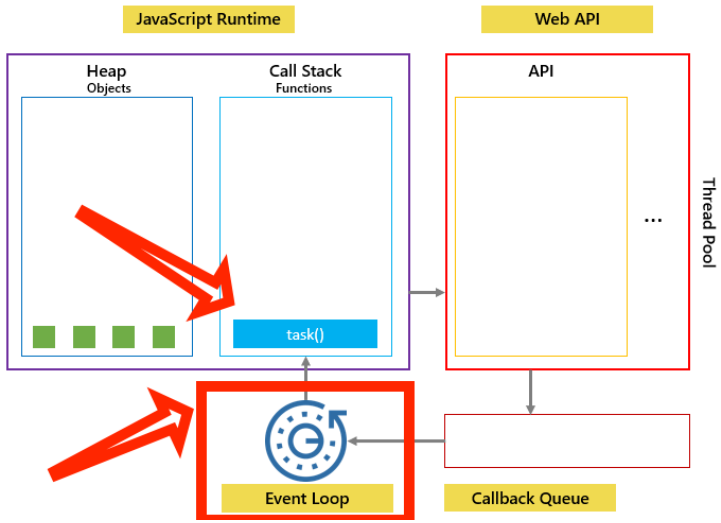
- When the time is out, the JavaScript engine places the task() function into a callback queue.



- The event loop is a constantly running process that monitors both the callback queue and the call stack until the call stack is empty.



- If the call stack is empty, the event loop places the next function, task(), from the callback queue to the call stack.



- As the setTimeout() function is expired immediately, the 2nd console.log() function (line 5) is placed in the call stack.
- As a result, the task() in the setTimeout() function is executed after 1000ms.

Code #

```
1 console.log('Hi!');
2 setTimeout(() => {
3     console.log('Execute immediately.');
4 }, 1000);
5 console.log('Bye!');
```

Command > Results

```
Hi!
Bye!
Execute immediately
    ↪ .
```

The Danger of Single Thread Model

setTimeout()

- When we use the setTimeout() function, we should be careful when we use it with the var modifier.
- In this example, we execute this code to expect the output is 1, 2, 3.

Code #

```
1 for (var i = 1; i <= 3; i++) {
2   setTimeout(function(){ console.log(i); }, 0);
3 };
```

Why not 1,2,3 but 4,4,4?

- When we execute this code, the output is not what we expect (1, 2, 3), but (4, 4, 4).
- However, we can understand these results as we know how call stack and event handler work.

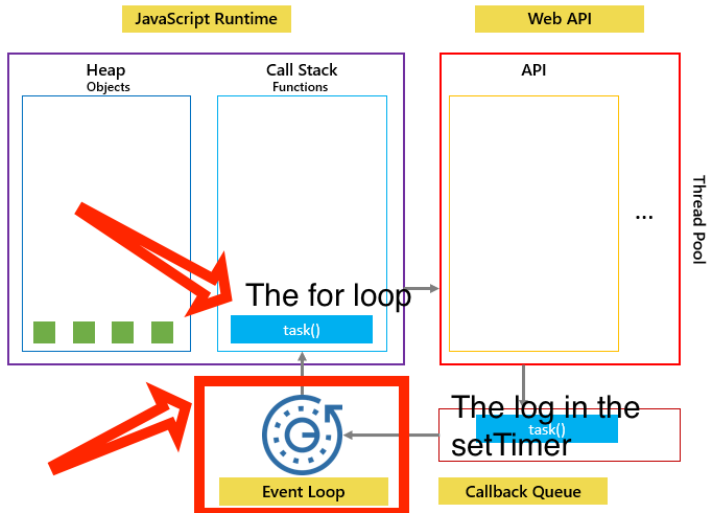
Code #

```
1 for (var i = 1; i <= 3; i++) {
2   setTimeout(function(){
3     console.log(i);
4   }, 0);
5 };
```

Command > Results

```
4
4
4
```

- It is because ‘JavaScript event handlers don’t run until the thread is free.‘
- In other words, because of its single-thread nature, the functions in the callback queue cannot be executed until the for loop is finished.



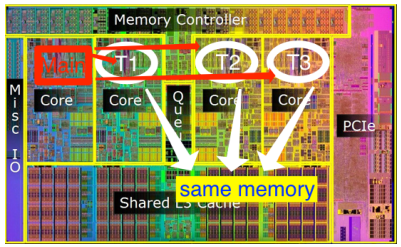
- And because of the nature of the var, the variable ‘i’ (with a value of 4) makes all the ‘console.log(i)’ code print out 4; the real code is roughly executed as follows.
- This is a tricky bug that is hard to understand unless we are familiar with the bad parts of JavaScript.

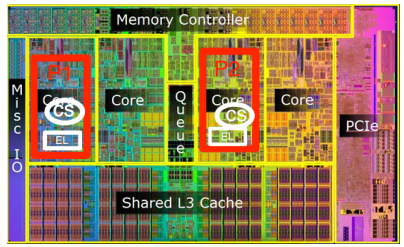
Code #

```
1 for (var i = 1; i <= 3; i++) {
2 };
3 // var i = 4; and this is executed three times
4 setTimeout(function(){ console.log(i); }, 0);
```

Performance Issue

- Compared to the multi-core programming that uses all the available cores effectively, the single-thread model is not as efficient.
- So, for computation-intensive programming, such as scientific programming, multi-thread programming is used.





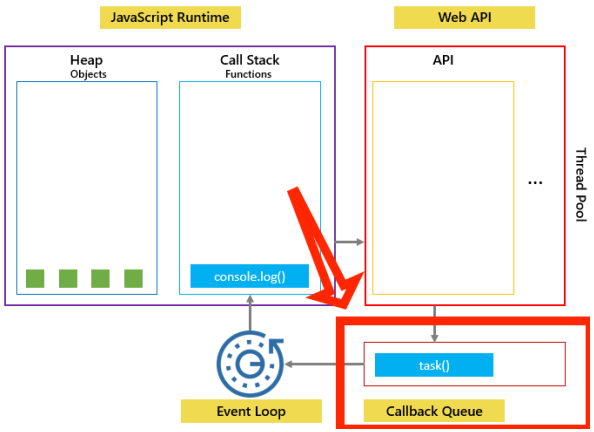
The Async and Nonblocking Functions

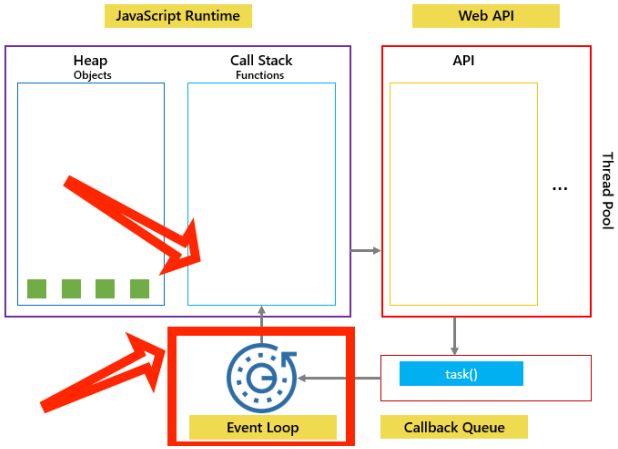
What are Async Functions?

- The term async function is a misnomer: if you call a function, your program simply won't continue until that function returns.
- A function "async" is that it can cause another function (called a callback when it's passed as an argument to the function) to run later, from the callback queue.

What is Nonblocking?

- Nonblocking means that async functions allow other functions to be executed by waiting in the thread pool stack and queue.





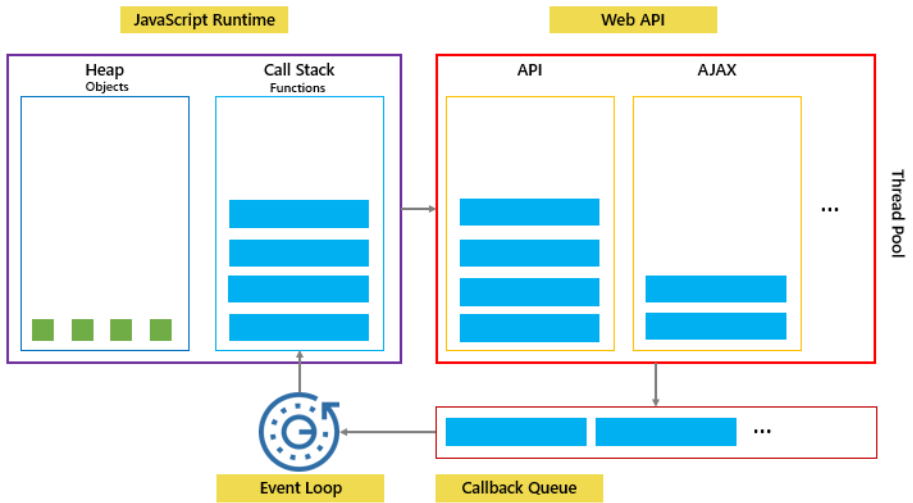
- Web servers can take advantage of this technology as they need to quickly process a high volume of incoming requests without wasting their time by just waiting and blocking everything.
- The callback functions that are waiting for any kind of request will be pushed into the *thread pool/callback queue* until the request is accepted and the call stack is empty.

Promises

Main Idea of Promises

- A Promise is an object that (a) represents a task with two possible outcomes (success or failure) and (b)holds callbacks that fire when one outcome or the other has occurred.

- We can understand that Promises are nothing more than *syntactic sugar* to use callback functions easily.
- JavaScript event loop manages the (a) callback queue and the (b) thread pool for promises automatically.



Promise in Node.js

- This code shows a template of a Node.js function that uses the Promise object.
- Notice that the Promise uses a callback function with two parameters: resolve and reject (they are also lambda expressions).

Code #

```
1 var a = new Promise((resolve, reject) => { ...
2   if (...) resolve(...)
3   else     reject(...)
4 }
5 another_function();
6 ...
```

- The Promise stores the lambda expression into the thread pool and executes the another_function().
- When there is a response from the server, the lambda expression is executed as it will be moved to the call stack.

Code #

```
1 var a = new Promise((resolve, reject) => {
2   ... // executed later
3 }
4 another_function(); // executed first
5 ...
```

- This is an example Node.js function getSumNum that promises to return the sum of two input values, a and b (it may take a while to get the result) at line 3.

Code #

```
1 function getSumNum(a, b) {
2   const customPromise = new Promise((resolve, reject) => {
3     const sum = a + b;
4     if(sum <= 5){ resolve(sum)
5     } else { reject(new Error('Oops!.. Number must be less than 5
        ↪ '))
6     }
7   })
8   return customPromise
9 }
```

- When the computation is finished, the result will be returned by invoking the resolve() function (when successful) or reject() function (when failed) at lines 4 – 6.

Code #

```
1 function getSumNum(a, b) {
2   const customPromise = new Promise((resolve, reject) => {
3     const sum = a + b;
4     if(sum <= 5){ resolve(sum)
5     } else { reject(new Error('Oops!.. Number must be less than 5
        ↪ '))
6     }
7   })
8   return customPromise
9 }
```

- The promise object that is returned (from getSumNum(1,3)) has the then() method that uses resolve(sum) in data (line 2).
- The catch has the catch() method that uses the new Error() object in err (line 7).

Code #

```
1 getSumNum(1, 3)
2 .then(data => {
3   console.log("initial data: " + data)
4   value = data + 1
5   console.log("received data: " + value)
6 })
7 .catch(err => {
8   console.log(err.message)
9 })
```

- Guess what the printed outputs are.

Code #

```
1  function getSumNum(a, b) {
2      const customPromise = new Promise((resolve, reject) => {
3          resolve(...); ...
4          reject(...) })
5      return customPromise
6  }
7  var sum = getSumNum(1, 3)
8  sum.then(data => {
9    console.log("initial data: " + data)
10   value = data + 1
11   console.log("received data: " + value)
12 })
13 .catch(err => { console.log(err) })
14 console.log("After the sum")
15 var sum = getSumNum(3, 3)
16 sum.then(data => { ... })
17     .catch(err => console.log(err.message)})
18 console.log("After the sum 2")
```

- Line 7 is executed, and as it returns a Promise object, the object is in the thread pool/callback queue.
- Then line 14 is executed.
- Likewise, line 15 is executed, and the Promise object is in the thread pool/callback queue.
- Then line 18 is executed.

- When the computation is over, either then or catch block is executed depending on which one, resolve() or reject(), is used to return the value.
- So this is the output.

```
Command >
Results
After the sum
After the sum 2
initial data: 4
received data: 5
Oops!.. Number must
    ↪ be less
    ↪ than 5
```

Async/Await

Async

- Using the Promise object with then() and catch() methods by manipulating lambda expression is not simple, so we have a better way to do the same thing with async/await.

Code #

```
1 function getSumNum(a, b) {
2     const customPromise = new Promise((resolve, reject) =>
3       {resolve(...); reject(...)})
4     return customPromise
5 }
6 var sum = getSumNum(1, 3)
7 sum.then(data => { ... })
8 .catch(err => { console.log(err) })
```

- Async function means the function returns a promise object automatically.
- In this example, we can make the getSumNum() function simple with async.

Code #

```
1  function getSumNum(a, b) {
2    const customPromise =
3      new Promise((resolve,
           ↪ reject) => {
4        const sum = a + b;
5        if(sum <= 5){
6          resolve(sum)
7        } else {
8          reject(new Error('...
               ↪ ')); }
9      })
10   return customPromise
11 }
```

Code #

```
1 async function getSumNum(a,b)
2 {
3   const sum = a + b;
4   if (sum <= 5) return sum
5   else throw new Error('...')
6 }
```

Await

- We can use the 'await' keyword to remove the usage of the 'then' method and a lambda expression with a try/catch.

Code #

```
1 getSumNum(1, 3)
2 .then(data => { ... })
3 .catch(err => { ... })
```

Code #

```
1 try {
2     var sum = await getSumNum
          ↪ (a, b)
3     ...
4 } catch (err) {
5     ...
6 }
```

- The code that uses await should be in an async function.
- So, the previous function is written as follows.

Code #

```
async function f(a, b) {
    try {
        var sum = await getSumNum(a, b)
        console.log("initial data: " + sum)
        value = sum + 1
        console.log("received data: " + value)
    } catch (err) {
        console.log(err.message)
    }
}
```

- Or, we can use the lambda expression.
- Be careful to append ';' at the end of the lambda expression to make it a statement.

Code #

```
1  (async () => {
2      try {
3          var sum = await getSumNum(2, 1)
4          console.log("initial data: " + sum)
5          value = sum + 1
6          console.log("received data: " + value)
7      } catch (err) {
8          console.log(err.message)
9      }
10 })(); // make a statement
11 console.log(...) // error without the previous ';'
```

Code

```
1  async function getSumNum(a, b) {
2    const sum = a + b;
3    if (sum <= 5) return sum;
4    else throw new Error('Oops!..
       ↪ Number must be less than 5')
5  }
6  async function f(a, b) {
7    try {
8      var sum = await getSumNum(a, b)
9      console.log("initial data: " +
         ↪ sum)
10     value = sum + 1
11     console.log("received data: " +
         ↪ value)
12   } catch (err) {
13     console.log(err.message)
14   }
15 }
16 f(1, 3)
17 console.log("After the sum")
18 f(3, 3)
19 console.log("After the sum 2")
```

Command > Results

```
After the sum
After the sum 2
initial data: 4
received data: 5
Oops!.. Number must be
    ↪ less than 5
```

Nested Promises

- We can nest Promises, and we can write the same code using async/await.

Code #

```
1  const getHen = () => {
2      return new Promise((resolve, reject) => {
3          setTimeout(() => resolve('Hen'), 100);
4      });
5  };
6  const getEgg = (hen) => {
7      return new Promise((resolve, reject) => {
8          setTimeout(() => resolve(`${hen} -> Egg`), 100);
9      });
10 };
11 const getCook = (egg) => {
12     return new Promise ((resolve, reject) => {
13         setTimeout (() => resolve(`${egg} -> Cook`), 100);
14     })
15 };
16 getHen().then(hen => getEgg(hen)).then(egg => getCook(egg))
17 .then(meal => console.log(meal));
```

Code

```
1  function delay(ms) {
2      return new Promise(resolve => setTimeout(resolve, ms));
3  }
4  const getHen = async () => {
5      await delay(100); return 'Hen'
6  }
7  const getEgg = async (hen) => {
8      await delay(100); return `${hen} -> Egg`
9  }
10 const getCook = async (egg) => {
11     delay(100); return `${egg} -> Cook`
12 }
13 async function f() {
14     try {
15         var hen = await getHen()
16         var egg = await getEgg(hen)
17         var cook = await getCook(egg)
18         console.log(cook)
19     } catch (err) {
20         console.log(err.message)
21     } finally {
22         console.log("Done")
23     }
24 }
25 f()
```

- When we have only one line in the body of a lambda expression, we can remove the block ({}) and 'return' keyword.

Code #

```
1 const getHen = () =>
2     new Promise((resolve, reject) => {
3         setTimeout(() => resolve('Hen'), 100);
4     })
```

- We can use 'reject()' to raise an error that is caught by the catch() function.

Code #

```
1  const getHen = () => ...
2  const getEgg = (hen) =>
3    new Promise((resolve, reject) => {
4      setTimeout(() => reject(new Error(`Error! ${hen} -> Egg`),
           ↪ 100));
5    })
6
7  const getCook = (egg) => ...
8  getHen().then(getEgg).then(getCook).then(console.log)
9  .catch(err => console.log(err.message));
```

- We can make the same code using async/await.

Code #

```
1  const getHen = async () => 'Hen'
2  const getEgg = async (hen) =>
3      {throw new Error(`Error! ${hen} -> Egg`)}
4  const getCook = async (egg) => `${egg} -> Cook`
5
6  async function f() {
7      try {
8          var hen = await getHen()
9          var egg = await getEgg(hen)
10         var cook = await getCook(egg)
11         console.log(cook)
12     } catch (err) {
13         console.log(`Something is wrong: ${err.message}`)
14     } finally {
15         console.log("Done")
16     }
17 }
18 f()
```

- When we need to catch the error and process it, we can use the catch() function.

Code #

```
1 const getHen = () => ...
2 const getEgg = (hen) => ...   reject(new Error(...))
3 const getCook = (egg) => ...
4
5 getHen()
6    .then(getEgg).catch(err => 'Bread')
7    .then(getCook).then(console.log)
8    .catch(err => console.log(err.message));
```

Code

```
1  const getHen = async () => 'Hen'
2  const getEgg = async (hen) => {
3    throw new Error(`Error! ${hen} -> Egg`)
4  }
5  const getCook = async (egg) => `${egg} -> Cook`
6
7  async function f() {
8      try {
9          var hen = await getHen()
10         try {
11             var egg = await getEgg(hen)
12         }
13         catch (err) {
14             egg = 'Bread'
15         }
16         var cook = await getCook(egg)
17         console.log(cook)
18     } catch (err) {
19         console.log(`Something is wrong: ${err.message}`)
20     } finally {
21         console.log("Done")
22     }
23 }
24 f()
```

- When we need to run async functions in parallel, we can use the Promise.all() function (line 10).

Code #

```
1  async function getApple() {
2    await delay(1000); return 'Apple';
3  }
4  async function getBanana() {
5    await delay(1000); return 'Banana';
6  }
7
8  async function pickFruits() {
9    const apple = getApple(); const banana = getBanana();
10   const fruits = await Promise.all([apple, banana]);
11   return fruits.join(' + ');
12 }
13 pickFruits().then(console.log);
```