JS Promise

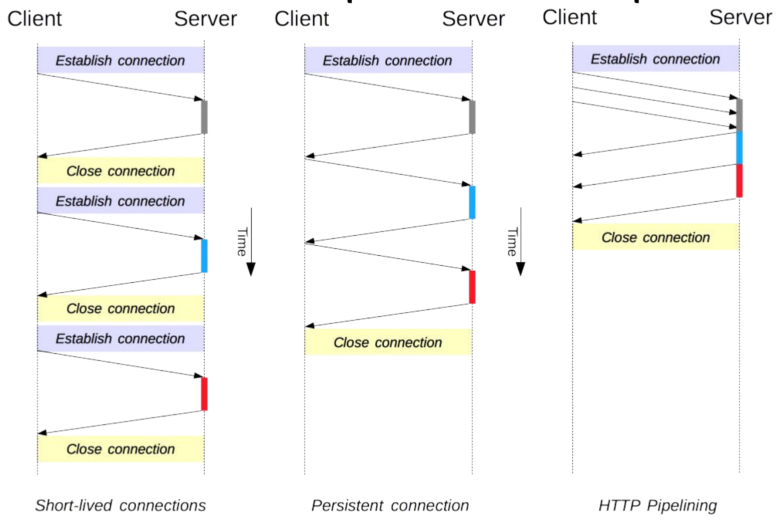
Handling Asynchronous Results

Topics Covered

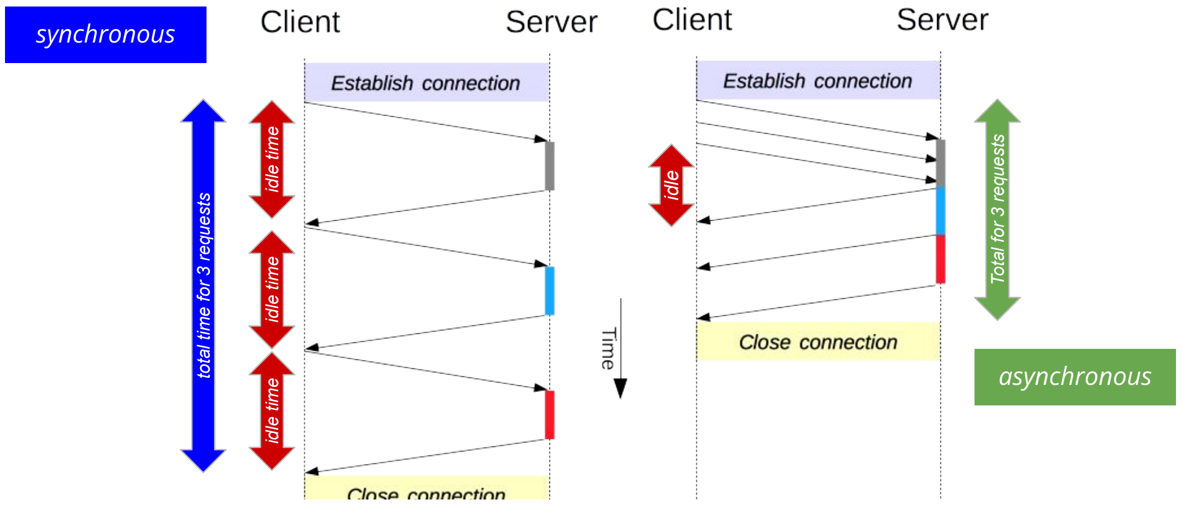
- Client/Server Communication
    - Synchronous
    - Asynchronous
- Callback functions (for handling asynchronous events)
- Promise

Reference: Promise Documentation (@ MDN)

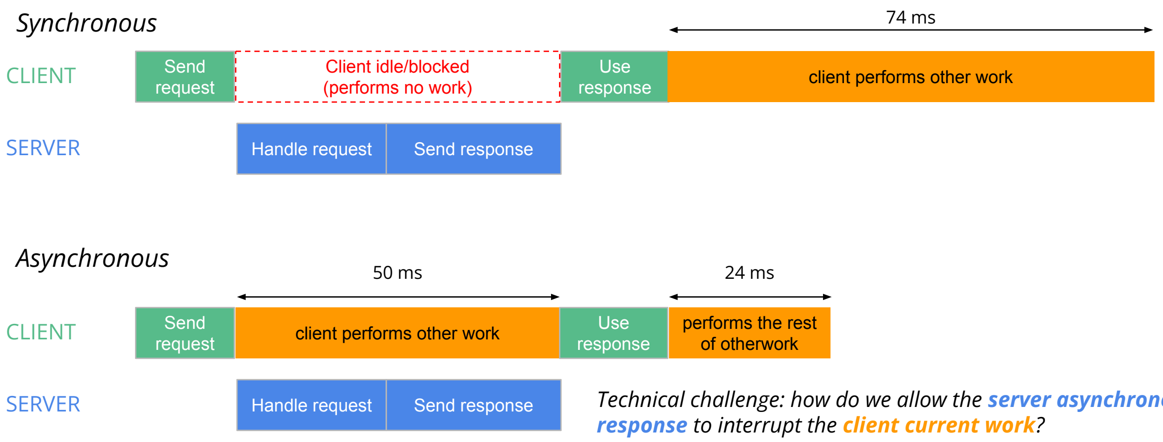
Client/Server: HTTP Requests & Responses



Client/Server: HTTP Requests & Responses



Synchronous vs. Asynchronous Requests



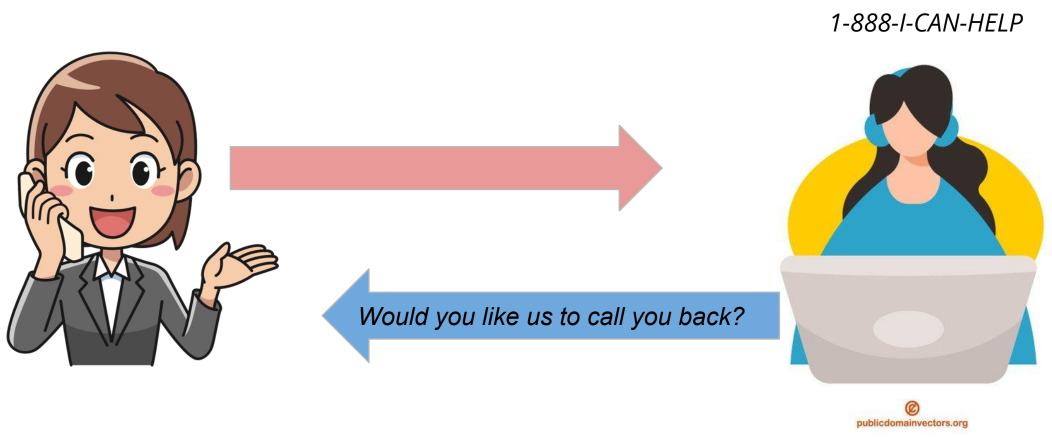
Technical challenge: how do we allow the ***server asynchronous response*** *to interrupt the* ***client current work****?*

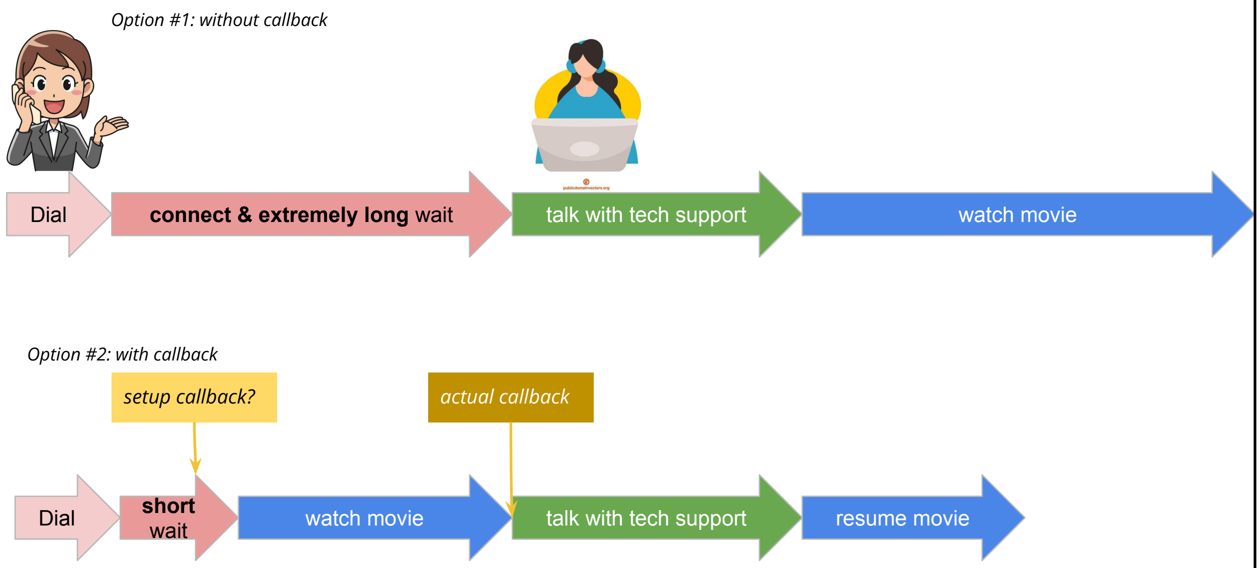
Sending Requests: *easy*

Receiving Async Responses: *requires extra setup*

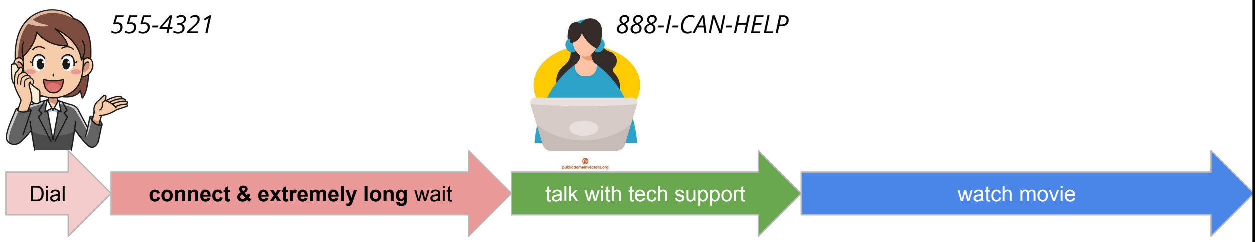
Callback Actions
(JS Callback Functions)

You are number 17 in line.....





Synchronous Call (in code)

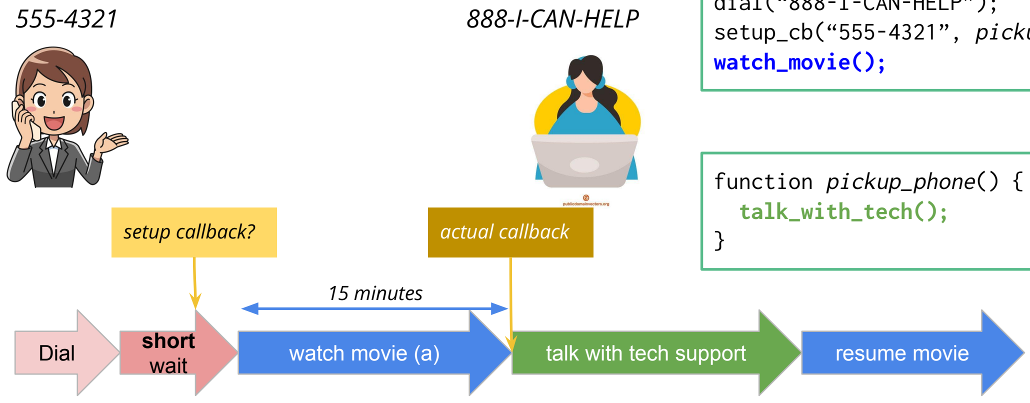


```
dial("888-I-CAN-HELP");
connect_and_long_wait();
talk_with_tech();
watch_movie();
```

Async: "out-of-order" execution
(Order of execution ≠ order of line of code)

Async Phone Calls with Callback (in code)

Asynchronous (incoming call) while you're watching movie



```
dial(“888-I-CAN-HELP”);
setup_cb(“555-4321”, pickup_phone);
watch_movie();
```

```
function pickup_phone() {
  talk_with_tech();
}
```

15 mins later

Callback fns (Fat Arrow)

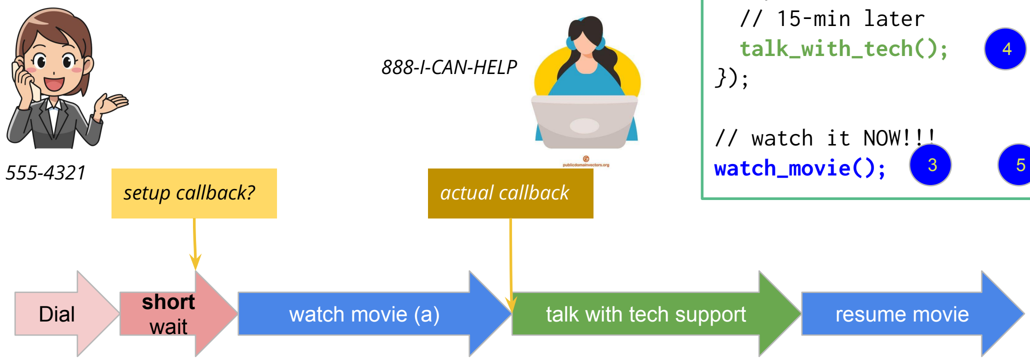
```
function pickup_phone() {
  talk_with_tech();
}

dial(“888-I-CAN-HELP”);
setup_cb(“555-4321”, pickup_phone);
watch_movie();
```

named function

```
dial(“888-I-CAN-HELP”);          1
setup_cb(“555-4321”, () => {     2
  // 15 min later
  talk_with_tech();             4
});
watch_movie();  3  5
```

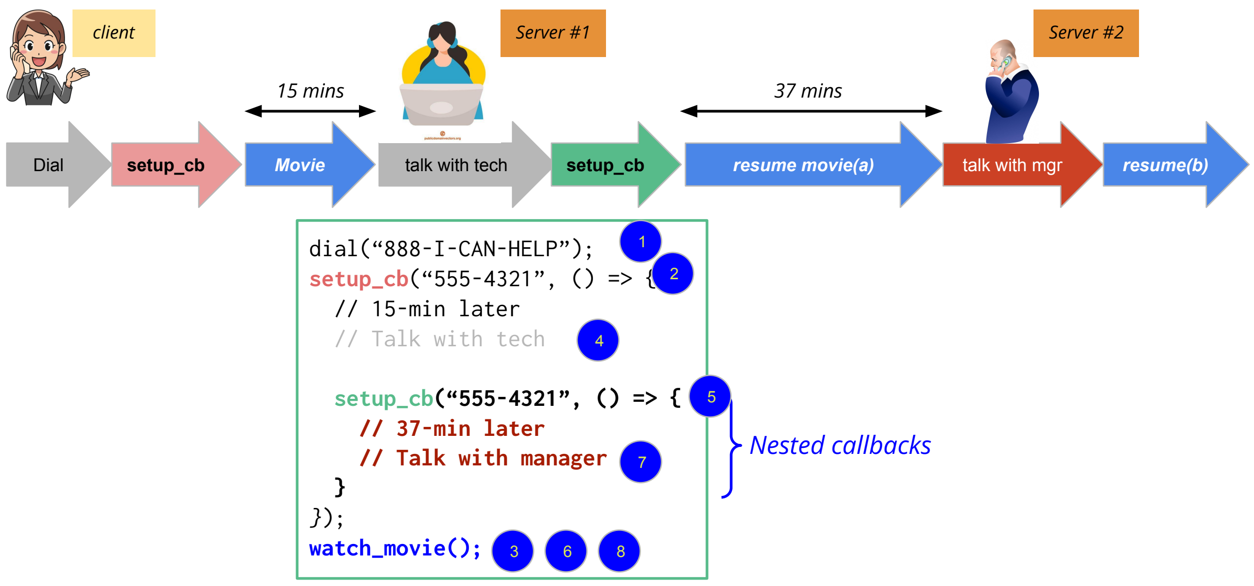
Fat arrow

Async: order of execution ≠ order of line of code



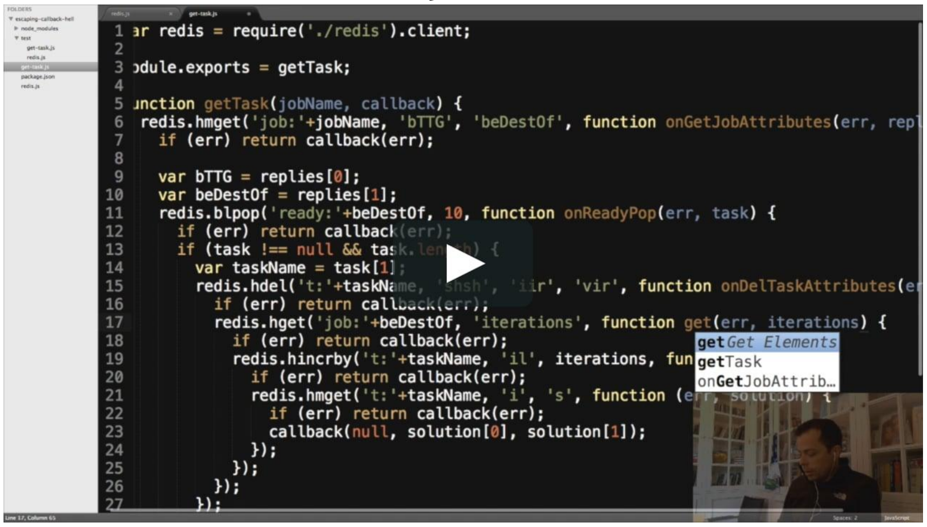
```
// start dialing ...
dial(“888-I-CAN-HELP”);         1

// call me back @ 555-4321
// then hangup to watch movie
setup_cb(“555-4321”, () => {     2
  // 15-min later
  talk_with_tech();         4
});

// watch it NOW!!!
watch_movie();    3      5
```

Tech: “But, you have to talk with my manager” (Nested Callback)



```
dial("888-I-CAN-HELP");
setup_cb("555-4321", () => {
  // 15-min later
  // Talk with tech

  setup_cb("555-4321", () => {
    // 37-min later
    // Talk with manager
  }
});
watch_movie();
```

Avoid Callback hell with JS Promise



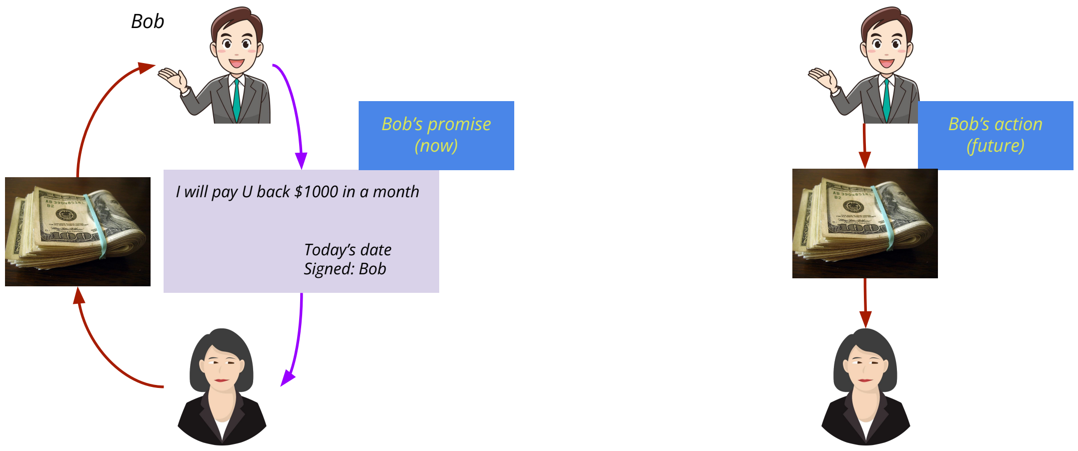
```
ar redis = require('./redis').client;

odule.exports = getTask;

unction getTask(jobName, callback) {
 redis.hmget('job:'+jobName, 'bTTG', 'beDestOf', function onGetJobAttributes(err, rep
   if (err) return callback(err);

   var bTTG = replies[0];
   var beDestOf = replies[1];
   redis.blpop('ready:'+beDestOf, 10, function onReadyPop(err, task) {
     if (err) return callback(err);
     if (task !== null && task.length) {
       var taskName = task[1];
       redis.hdel('t:'+taskName, 'shsh', 'iir', 'vir', function onDelTaskAttributes(e
         if (err) return callback(err);
         redis.hget('job:'+beDestOf, 'iterations', function get(err, iterations) {
           if (err) return callback(err);
           redis.hincrby('t:'+taskName, 'il', iterations, fun
             if (err) return callback(err);
             redis.hmget('t:'+taskName, 'i', 's', function (err, solution) {
               if (err) return callback(err);
               callback(null, solution[0], solution[1]);
             });
           });
         });
       });
```

How to Initiate Async HTTP Requests?

- `fetch()` function
  - Native in browser
  - NPM `node-fetch`
- Axios library
- Both fetch() and axios() use JS Promise

IOU = I owe you note
Promise to pay debt/loan

Borrowing Money: Promise Now, Pay Later



Today (now): Beth receives a promise

1 month later (future): Beth receives cash

A promise = *now* confirmation of *future* action(s)

A JS promise = a "***now***" object representing data which will become available in the ***future***

Promise Example

```
function nthPrime(nth: number): Promise<number> {
  // work takes 10 seconds
  return Promise.resolve(_______);
}
```

```
function nthPrimeNow(nth: number): number {
  // work takes 10 seconds
  return _______;
}
```

```
console.log(“Start”);
const prom = nthPrime(500);
prom.then ((pr: number) => {
    console.log(“The 500th prime is”, pr);
  });
doMoreWork();
```

```
console.log(“Start”);
const pr = nthPrimeNow(500);
console.log(“The 500th prime is”, pr);
doMoreWork();
```

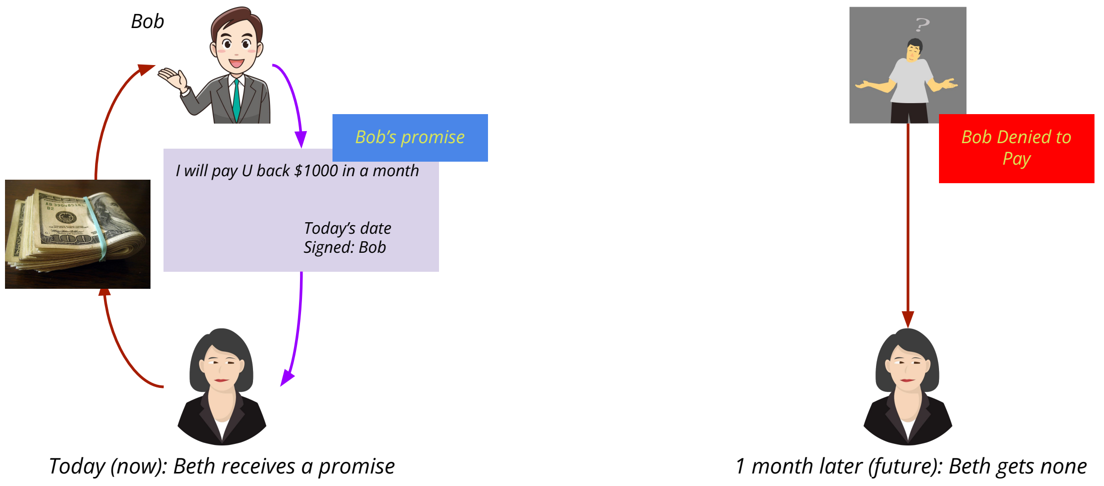
Compare the order of execution

```
Start
Partial output of doMoreWork()

# After 10 seconds
The 500th prime is 3571
More output from doMoreWork()
```

```
Start

# After 10 seconds
The 500th prime is 3571
Output of doMoreWork()
```

Loan is either paid-off or defaulted
Promise is either resolved or rejected

Borrowing Money: Promise Now, Never Pay


Bob
Bob's promise
I will pay U back $1000 in a month
Today's date
Signed: Bob
Bob Denied to Pay
Today (now): Beth receives a promise
1 month later (future): Beth gets none

Promise settlement: resolve() or reject()

```
function nthPrime(nth: number): Promise<number> {
  if (nth < 100_000) {
    // assume prime calculation takes 10 seconds
    return Promise.resolve(a_prime_number_here);
  }
  else
    return Promise.reject("Can't compute prime")
}
```

```
console.log("Start");
nthPrime(500)
  .then ((pr: number) {
     console.log("Prime is", pr);
   });
  .catch((err:any) => {
     console.log("Rejected", err);
   });
console.log("Here");
```

```
# Watch for order of execution
Start
Here

# if the promise is resolved
# After 10 seconds ...
Prime is 3571

# if the promise is rejected
Rejected Can't compute prime
```

Using JS Promise

- Basic methods: `then()`, `catch()`, `finally()`
- Basics static functions
  - `Promise.resolve()`
  - `Promise.reject()`
- Advanced (for handle *multiple concurrent* promises)
  - `Promise.all(`*`array`*`)`: wait until all the promises in the array are resolved
  - `Promise.allSettled(`*`array`*`)`: wait until all the promises in the array are either resolved or rejected
  - `Promise.any(`*`array`*`)`: wait until ONE of the promises in the array is resolved
  - `Promise.race(`*`array`*`)`: wait until ONE of the promises in the array is either resolved or rejected

`then`-able chains

Then and then and then and …

```
function nthPrime(nth: number): Promise<number> {
  // more code here
  return Promise.____;
}
```

```
function toRomanNumeral(inputNum: number): string {
  // conversion to Roman numberal
  return _______
}
```

Return from a then() *becomes a Promise to the next* then() *inline*

```
nthPrime(500)
  .then((p:number): string => {
    return toRomanNumeral(p);
  })
  .then((rome: string) => {
    console.log(`Prime in roman numeral ${rome}`);
  });
```

```
// After 1-line return elimination
nthPrime(500)
  .then((p:number): string => toRomanNumeral(p))
  .then((rome: string) => {
    console.log(`Prime in roman numeral ${rome}`);
  });
```

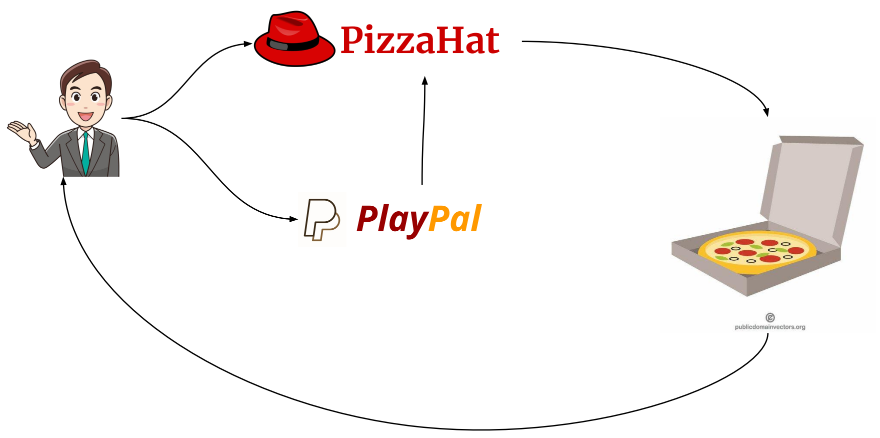
Then and then and … (promise “unpacked”)

```
function nthPrime(nth: number): Promise<number> {
  // more code here
  return Promise.____;
}
```

```
function promNum(inputNum: number): Promise<string>
{
  // conversion to Roman numberal
  return Promise._______;
}
```

```
nthPrime(500)
  .then((p:number): Promise<string> => promNum(p))
  .then((rome: string) => { // “unpacked”!!!
    console.log(`Prime in roman numeral ${rome}`);
  });
```

Online Pizza Order & 3rd party payment



Online Pizza Order (code setup)

```
function orderPizza(___): Promise<PizzaOrder> {
  return Promise.resolve(___);
}
```
PizzaHat

```
function makePizza(____): Promise<PizzaBox> {
  return Promise.resolve(___);
}
```
PizzaHat

```
function playWithPal(name: string, payAmt: number):
  Promise<ProofOfPlay> {
  return Promise.resolve(___);
}
```
PlayPal

```
type PizzaOrder = {
  crustStyle: “Classic” |
    “ThinCrust” |
    “HandTossed”;
  size: number;
  toppings: Array<Topping>;
  customerName: string;
  price: number
}
```

```
type PizzaBox = {
  customerName: string;
  inStorePickup: boolean
}

type ProofOfPlay = {
  payer: string;
  payee: string;
  amount: number;
  transactionDate: string
}
```

```
orderPizza(____)
  .then((ord: PizzaOrder) => playWithPal(__, __))
  .then((proof: ProofOfPlay) => makePizza(____))
  .then((box: PizzaBox) => {
    console.log(“Open the box and enjoy!”);
   })
  .catch((err:any) => {
    console.error(“Can’t complete order”);
  });
```

Online Pizza Order (chaining)

```
function orderPizza(___): Promise<PizzaOrder> {
  return Promise.resolve(___);
}
```

PizzaHat

```
function makePizza(____): Promise<PizzaBox> {
  return Promise.resolve(___);
}
```

PizzaHat

```
function playWithPal(name: string, payAmt: number):
  Promise<ProofOfPlay> {
  return Promise.resolve(___);
}
```

PlayPal

Promise: with finally

```
function nthPrime(int nth):
  Promise<number>
{
  // work takes 10 seconds
  return _______;
}
```

```
console.log("Start");
nthPrime(500)
  .then ((pr: number) {
    console.log("Prime is", pr);
  });

doMoreWork();
```

```
console.log("Start");
nthPrime(500)
  .then ((pr: number) {
    console.log("Prime is", pr);
  })
  .finally(() => {
    doMoreWork();
  });
```

```
Start
Partial output of doMoreWork()

# After 10 seconds
Prime is 3571
More output from doMoreWork()
```

```
Start


# After 10 seconds
Prime is 3571
Output of doMoreWork()
```

Promise: put them all together

```
work_with_promise(____, _____, ____)
  .then((arg: type1): type2 => {
    // more code here
    return ____;
  })
  .then((arg: type2): type3 => {
    // more code here
    return ____;
  })
  .then((arg: type2): type3 => {
    // more code here
    return ____;
  })
  /* more chain of .then here */
  .catch((err:any) => {
    // Error handling code here
  })
  .finally(() => {
    // Overall "cleanup" code here
  })
```

Any `Promise.reject()` here will be caught by

Promise.reject() skips then-chain until it finds a .catch

async & await

Async functions

```
function nthPrime(nth: number): Promise<number> {
  let thePrime:number;
  // more code here
  return Promise.resolve(thePrime);
}
```

```
async function nthPrime(nth: number): Promise<number> {
  let thePrime:number;
  // more code here
  return thePrime;    // Promise.resolve() is not required
}
```

```
const nthPrime = async (nth: number): Promise<number> => {
  let thePrime:number;
  // more code here
  return thePrime;    // Promise.resolve() is not required
}
```

await: rewrite in synchronous *style*

```
orderPizza(____)
  .then((ord: PizzaOrder) => playWithPal(__, __))
  .then((proof: ProofOfPlay) => makePizza(____))
  .then((box: PizzaBox) => {
    console.log("Open the box and enjoy!");
   })
  .catch((err:any) => {
    console.error("Can't complete order");
  });
```

Await can only be used inside async functions

```
const doPizza = async (): Promise<void> => {
  try {
    const ord  : PizzaOrder  = await orderPizza(____);
    const proof: ProofOfPlay = await playWithPal(__, __);
    const box  : PizzaBox    = await makePizza(____);
    console.log("Open the box and enjoy!");
  }
  catch((err:any) => {
    console.error("Can't complete order");
  });
}
```

Advanced Topics

Promise class: constructor

```
new Promise( function(resolvFn, rejectFn) { /* body */ } );
```

```
new Promise(                                          );
```

The promise constructor takes

```
function(                    ) { /* body */ }
```

a function as its input argument

```
(resolvFn, rejectFn)
```

that function takes 2 arguments (which are also a function)

- The first func resolves the promise
- The second func rejects the promise

Using Promise class

Static Function Shortcut	Promise constructor
`Promise.resolve("Hello World")`	`new Promise(resolve => {        resolve("Hello World"); });`
`Promise.resolve("Hello World")`	`new Promise(resolve => { return resolve("Hello World"); });`
`Promise.resolve("Hello World")`	`new Promise(resolve =>          resolve("Hello World")   );`
`Promise.reject("Nope")`	`new Promise((_, reject) =>      reject("Nope")           );`

Delayed Response

```
function delayedText (msg: string, delay: number): Promise<string> {
  setTimeout (() => {
    return Promise.resolve(msg);
  }, delay);
}
```

Failed attempt: missing return

```
function delayedText (msg: string, delay: number): Promise<string> {
  return setTimeout (() => {
    return Promise.resolve(msg);
  }, delay);
}
```

Failed attempt: incorrect return type

```
function delayedText (msg: string, delay: number): Promise<string> {
  return new Promise((resolve) => {
    setTimeout(() => resolve(msg), delay);
  });
}
```

Correct implementation