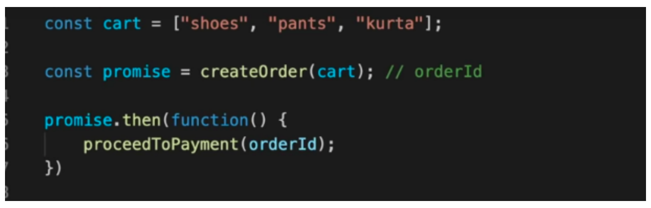
Creating a Promise, Chaining and Error Handling

Wednesday, 23 August 2023 1:24 AM

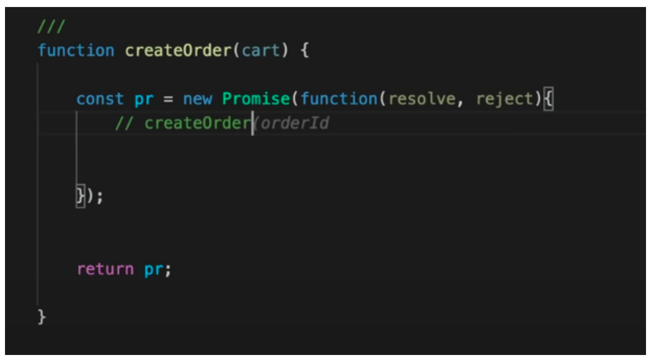
Previously we had seen how to consume a promise.



```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

promise.then(function() {
    proceedToPayment(orderId);
})
```

→ So createOrder() API returned a promise, and then we attach a callback f^n to this promise.

→ Now, we will create a promise which will be returned by createOrder().

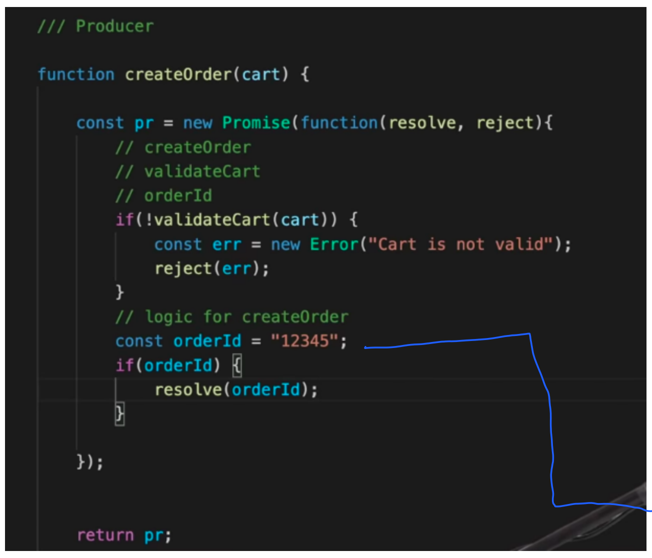


```
///
function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){
        // createOrder orderId


    });


    return pr;
}
```

→ To create a promise, we use new keyword and constructor f^n which takes in 2 parameters: resolve & reject

→ Inside the promise we will write the logic.

→ Fulfilled will be when promise returns orderId. Rejected will be when the promise throws an error.

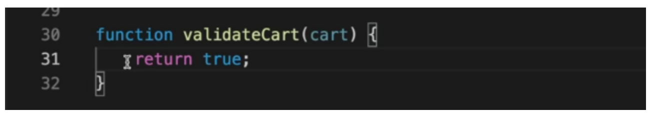


```
/// Producer

function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){
        // createOrder
        // validateCart
        // orderId
        if(!validateCart(cart)) {
            const err = new Error("Cart is not valid");
            reject(err);
        }
        // logic for createOrder
        const orderId = "12345";
        if(orderId) {
            resolve(orderId);
        }

    });


    return pr;
```

→ This is the promise producer f^n!

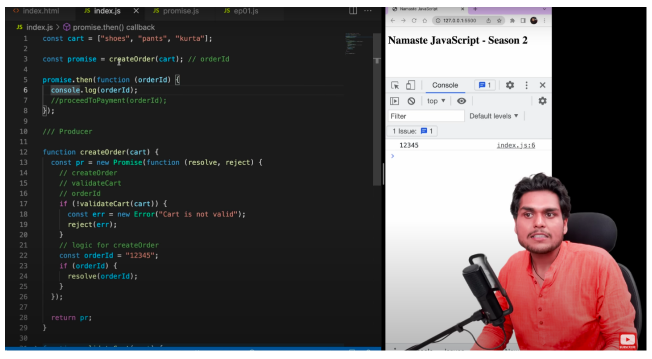
→ First we will see if the cart items are valid. We will run validateCart() f^n & if they are invalid we will throw an error & use the reject() f^n.

→ Assuming here we make a DB call & fetch the order ID and we get the orderId, so when we get the orderId we will run the resolve() f^n.



```
function validateCart(cart) {
    return true;
}
```

→ validateCart() f^n just for the sake of explanation.

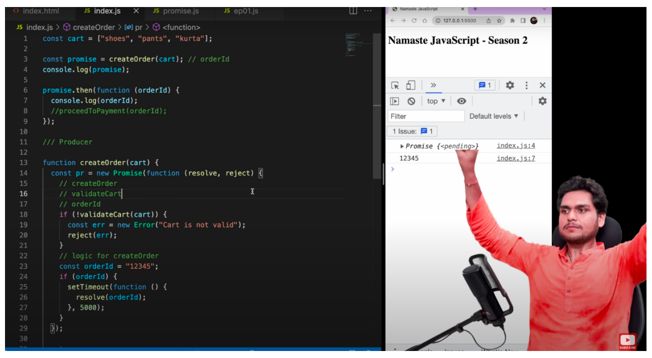


```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

promise.then(function (orderId) {
  console.log(orderId);
  //proceedToPayment(orderId);
});

/// Producer

function createOrder(cart) {
  const pr = new Promise(function (resolve, reject) {
    // createOrder
    // validateCart
    // orderId
    if (!validateCart(cart)) {
      const err = new Error("Cart is not valid");
      reject(err);
    }
    // logic for createOrder
    const orderId = "12345";
    if (orderId) {
      resolve(orderId);
    }
  });

  return pr;
}
```

```
12345                index.js:6
```

→ So basically createOrder() returns a promise and since cart items are present & validateCart() returns true, promise gets resolved.

→ Now, when promise gets resolved, our attached callback f^n gets the returned orderId & consoles it.



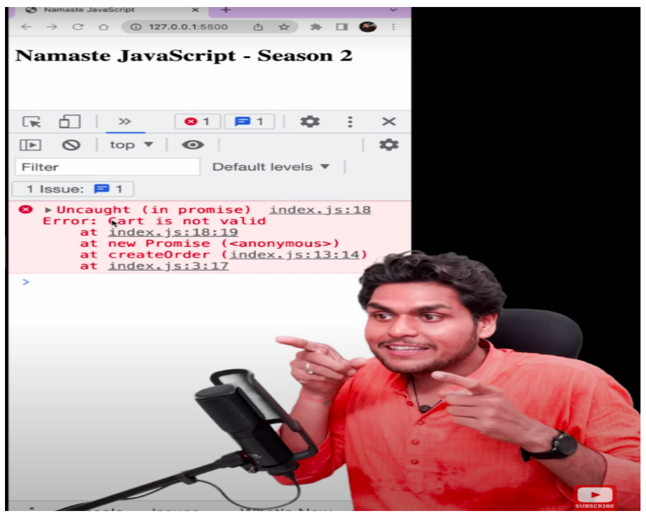
```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId
console.log(promise);

promise.then(function (orderId) {
  console.log(orderId);
  //proceedToPayment(orderId);
});

/// Producer

function createOrder(cart) {
  const pr = new Promise(function (resolve, reject) {
    // createOrder
    // validateCart
    // orderId
    if (!validateCart(cart)) {
      const err = new Error("Cart is not valid");
      reject(err);
    }
    // logic for createOrder
    const orderId = "12345";
    if (orderId) {
      setTimeout(function () {
        resolve(orderId);
      }, 5000);
    }
  });
```

Namaste JavaScript - Season 2

```
▸ Promise {<pending>}    index.js:4
12345                    index.js:7
```

→ If we put a delay in the producer f^n, then firstly pending promise is printed because JS does not wait for anyone & consoles the promise, but as soon as the promise is resolved event loop pushes it into the call stack & orderId is printed.

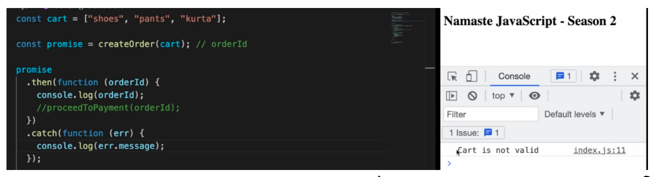
→ If promise returns rejected state & we haven't handled the rejected state then we get an error:



Namaste JavaScript - Season 2

```
⊗ ▸Uncaught (in promise)   index.js:18
  Error: Cart is not valid
      at index.js:18:19
      at new Promise (<anonymous>)
      at createOrder (index.js:13:14)
      at index.js:3:17
```

→ In order to handle rejection of promise, we must do error handling, which is done with .catch() .



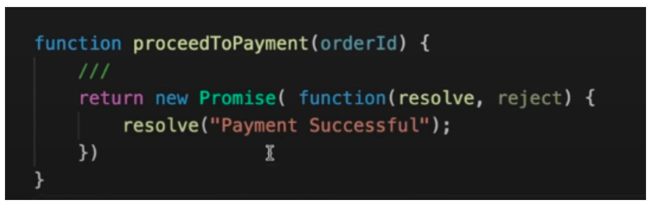
```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId

promise
  .then(function (orderId) {
    console.log(orderId);
    //proceedToPayment(orderId);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

Namaste JavaScript - Season 2

```
Cart is not valid    index.js:11
```

→ Thus, we can attach a failure callback f^n & handle the rejection state of ~~the~~ promise.

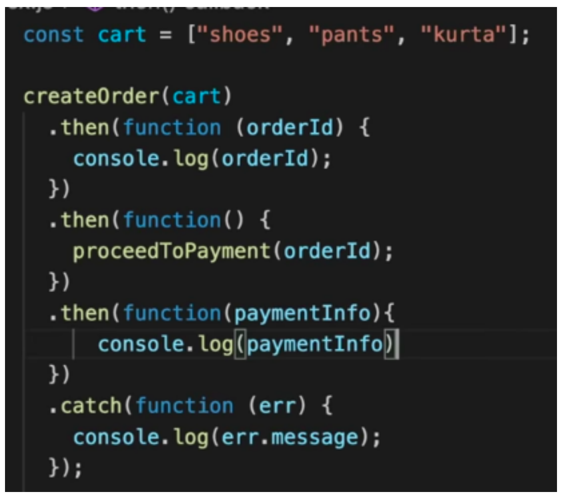
→ Promise chaining:



```
function proceedToPayment(orderId) {
    ///
    return new Promise( function(resolve, reject) {
        resolve("Payment Successful");
    })
}
```

Lets assume there is another promise that takes in order Id.

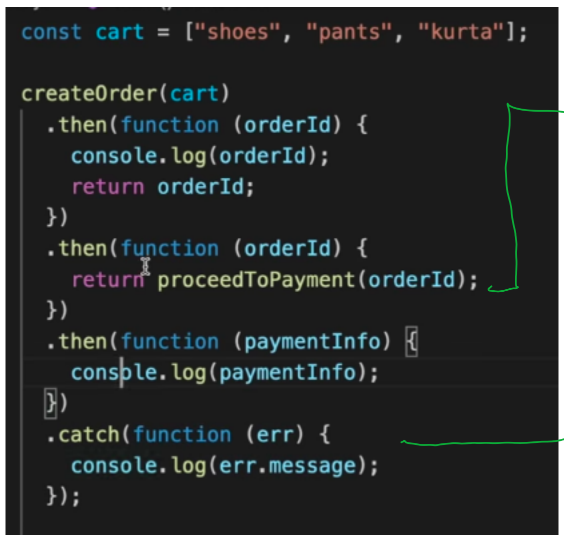
Since this depends on the result of createOrder(), so we must do promise chaining.



```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
  })
  .then(function() {
    proceedToPayment(orderId);
  })
  .then(function(paymentInfo){
    console.log(paymentInfo)
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

→ We can make a promise chain like this.
→ So whatever is returned from ProceedToPayment() will automatically be passed to the next attached callback f^n when the previous one is resolved.

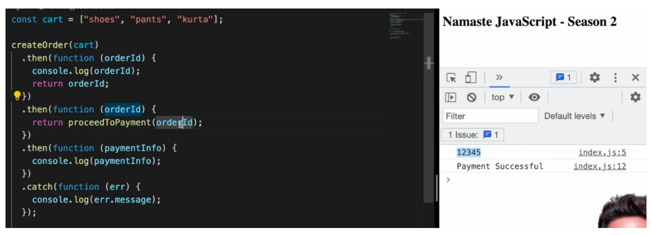
→ But we must return in each step of the promise chain, so the right code would be :



```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

→ this would return us a promise which will be resolved then paymentInfo will be resolved.

→ This .catch() handles any error encountered in whole of the promise chain.



```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })
  .catch(function (err) {
    console.log(err.message);
  });
```

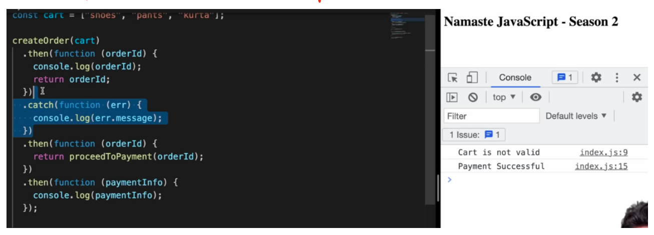
Namaste JavaScript - Season 2

```
12345                  index.js:5
Payment Successful     index.js:12
```

→ So this will be the o/p of the code.

→ So, when createorder () promise is resolved it returns `12345' & `12345' is consoled.
→ We return `12345' & pass it to the callback f^n which runs auto-matically because previous promise resolved, Now we pass orderID to new promise & when resolved it returns us the payment info.
→ when proceedToPayment promise is resolved our next callback f^n runs & payment info is consoled.

→ What if we put .catch() before other .then()?



```
const cart = ["shoes", "pants", "kurta"];

createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId;
  })
  .catch(function (err) {
    console.log(err.message);
  })
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  });
```

Namaste JavaScript - Season 2

```
Cart is not valid      index.js:9
Payment Successful     index.js:15
```

→ if this happens then all the other .then() after .catch() will most definitely be called.