JS Promises

by Minh

Asynchronous programming

```
var x = null

setTimeout(function () {
  x = 100
}, 1000)

console.log(x)
```

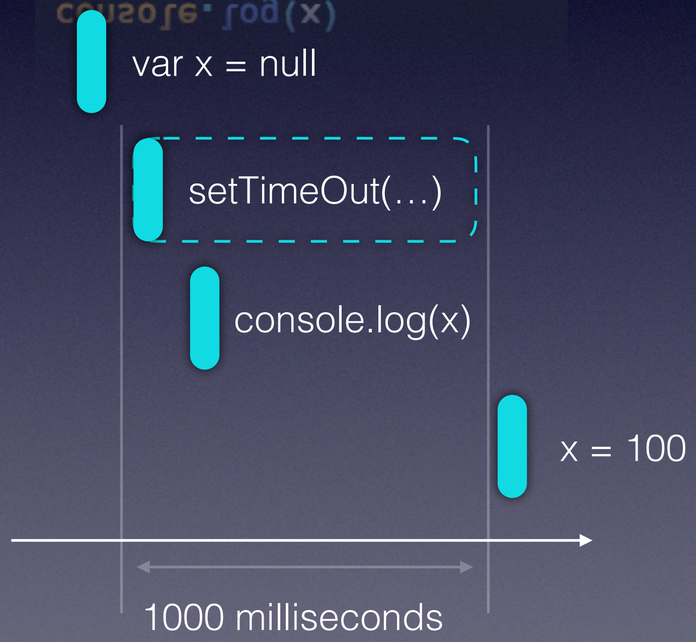
```
var x = null

setTimeout(function () {
  x = 100
  console.log(x)
}, 1000)
```

Asynchrony, in computer programming, refers to the occurrence of events independently of the main program flow and ways to deal with such events.

- Wikipedia, 2017 -

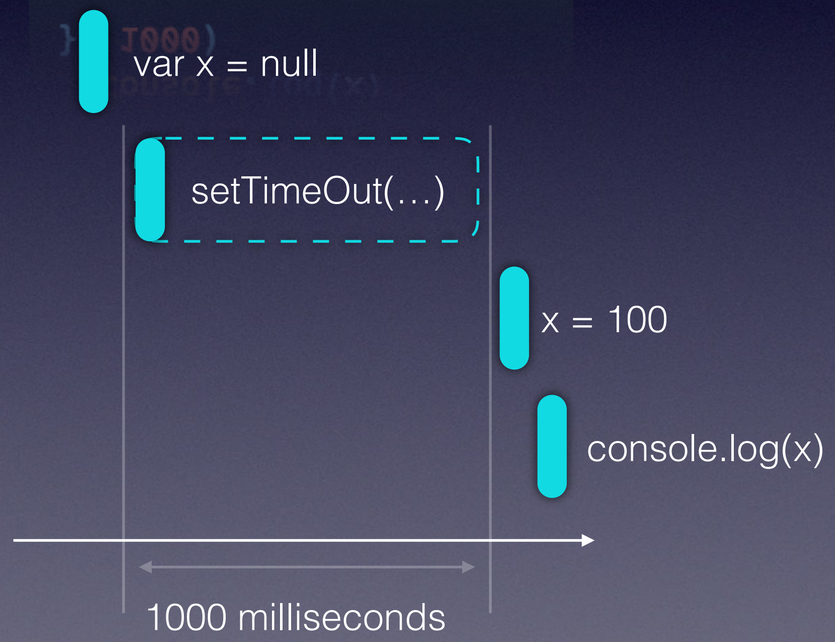
Asynchronous programming

```
var x = null

setTimeout(function () {
  x = 100
}, 1000)

console.log(x)
```



output: null

```
var x = null

setTimeout(function () {
  x = 100
  console.log(x)
}, 1000)
```



output: 100

Asynchronous programming - callback

In computer programming, callback is **any executable code that is passed as an argument to other code**, which is expected to call back (execute) the argument at a given time.

- Wikipedia, 2017 -

```
var x = null

setTimeout(function () {
  x = 100
}, 1000)

console.log(x)
```

1 Asynchrony: the event of hitting 1 second (or 1000 milliseconds) after the call to setTimeout() is independent of the main program flow

2 Javascript does not wait until 1 second later to execute the console.log()

3
```
function(){
    x = 100
}
```

This is the **“callback” function passed to the setTimeout function as an argument** that javascript executes when the event of hitting 1 second after setTimeout() is called happens.

```
var x = null

setTimeout(function () {
  x = 100
  console.log(x)
}, 1000)

function(){
  x = 100
  console.log(x)
}
```

The **console.log** needs to be included inside the callback back function for it to be delayed

Callback: the problem

```
// code to get all the product models a customer has ordered
const productsOrdered = [];
getCustomerById(2, function(customer){
  getOrdersByIds(customer.orders, function(orders){
    orders.forEach(function(order){
      getModelById(order.prod_id, function(product){
        productsOrdered.push(product);
        if(orders.indexOf(order) === orders.length - 1){
          const product_models = productsOrdered.map(function(prod){ return prod.model });
          console.log( "Customer name " + customer.name + " has ordered " + product_models )
        }
      })
    })
  })
})
})
  })
    })
```

CALLBACK HELL

Infinitely nested function calls

Hard to read, hard to reason

Prone to bugs

Promises: the solution

```
var _customer = null;
getCustomerById(2)
  .then(function(customer){
    _customer = customer
    return getOrdersByIds(customer.orders)
  })
  .then(function(orders){
    return Promise.all(orders.map(function(order){
      return getModelById(order.prod_id)
    }))
  })
  .then(function(products){
    const product_models = products.map(function(prod){ return prod.model });
    console.log( "Customer name " + _customer.name + " has ordered " + product_models )
  })
```

CHAINED PROMISES

Flattened step-by-step syntax and logic

Easier to read and reason about

Easier to debug

The promise flow

Promise construction → Resolve / reject → Outside the promise

```
function getCustomerById(cus_id){
  const customer = dataBase.customer.filter(function(cus){
    return cus.id === cus_id;
  })[0];
  return new Promise(function(resolve, reject) {
    setTimeout(function () {
      if(customer){
        resolve(customer);
      } else {
        reject("NO_CUSTOMER_FOUND");
      }
    }, 500);
  });
}
```

return a new promise object, which **takes a callback function** that performs an **asynchronous operation**

the asynchronous operation will eventually either **“resolve”** or **“reject”** the Promise **with the result of the operation**

The promise flow

Promise construction → Resolve / reject → Outside the promise

```
function getCustomerById(cus_id){
  const customer = dataBase.customer.filter(function(cus){
    return cus.id === cus_id;
  })[0];
  return new Promise(function(resolve, reject) {
    setTimeout(function () {
      if(customer){
        resolve(customer);
      } else {
        reject("NO_CUSTOMER_FOUND");
      }
    }, 500);
  });
}
```

resolve: indicates that the promise has success successfully been fulfilled and provides expected output

reject: indicates that the promise could not be fulfilled and provides error data for error handling.

*If a customer with such ID is found, **resolve** the promise and provide the customer.*
*Else, **reject** the promise and give the reason why.*

> *Here's my promise: I will **resolve** this issue when I can, **then** you can move on with your work given my resolution. If the issue cannot be resolved, I will **reject** the situation and you can **catch** my error and try something else.*

The promise flow

Promise construction > Resolve / reject > Outside the promise

```
getCustomerById(233)
  .then(function(customer){
    _customer = customer
    return getOrdersByIds(customer.orders)
  })
  .catch(function(err){
    console.log(err);
  })
```

return new Promise(function(){…})

.then() - Promise.prototype.then

- **receives a callback** to call **when the the Promise is resolved**
- callback receives the value resolved from the promise
- returns a new Promise object

.catch() - Promise.prototype.catch

- **catches errors reported by any reject()**
- callback receives the error

Chaining promises

```
var _customer = null;
getCustomerById(2)
  .then(function(customer){
    _customer = customer
    return getOrdersByIds(customer.orders)
  })
  .then(function(orders){
    return Promise.all(orders.map(function(order){
      return getModelById(order.prod_id)
    }))
  })
  .then(function(products){
    const product_models = products.map(function(prod){ return prod.model });
    console.log( "Customer name " + _customer.name + " has ordered " + product_models
  })
```

Asynchronous logic in synchronous-like style

Easier to read and reason about

Easier to debug

Chaining promises - Error handling

```
getCustomerById(233)
  .then(function(customer){
    _customer = customer
    return getOrdersByIds(customer.orders)
  })
  .then(function(orders){
    return Promise.all(orders.map(function(order){
      return getModelById(order.prod_id)
    }))
  })
  .then(function(products){
    const product_models = products.map(function(prod){ return prod.model });
    console.log( "Customer name " + _customer.name + " has ordered " + product_models )
  })
  .catch(function(err){
    if(err === "NO_CUSTOMER_FOUND"){
      console.log("No customer was found");
    } else {
      console.log("An unknown error happened");
    }
  })
```

one .catch() to catch any error thrown by any chained promise

Useful methods - `Promise.all()`

```
getCustomerById(3)
  .then(function(customer){
    _customer = customer
    return getOrdersByIds(customer.orders)
  })
  .then(function(orders){
    return Promise.all(orders.map(function(order){
      return getModelById(order.prod_id)
    }))
  })
  .then(function(products){
    console.log('got products:', JSON.stringify(products));
  })
```

`Promise.all(iterable)`

- **arguments**: `Array<Promises>`
- **return**: `new Promise()` that resolves when all of the promises in the argument array have resolved

- **.then** callback receives an array of all resolved value in corresponding order of the original promise array

Useful methods - `Promise.race()`

Server 1

```
const server1 = {
  loadTime: 500,
  getCustomerById: function(cus_id){
    const customer = dataBase.customer.filter(function(cus){
      return cus.id === cus_id;
    })[0];
    return new Promise(function(resolve, reject) {
      setTimeout(function () {
        if(customer){
          resolve(customer);
        } else {
          reject("NO_CUSTOMER_FOUND");
        }
      }, this.loadTime);
    });
  }
}
```

load time: 500ms

Server 2

```
const server2 = {
  loadTime: 300,
  getCustomerById: function(cus_id){
    const customer = dataBase.customer.filter(function(cus){
      return cus.id === cus_id;
    })[0];
    return new Promise(function(resolve, reject) {
      setTimeout(function () {
        if(customer){
          resolve(customer);
        } else {
          reject("NO_CUSTOMER_FOUND");
        }
      }, this.loadTime);
    });
  }
}
```

load time: 300ms

Server 3

```
const server3 = {
  loadTime: 1000,
  getCustomerById: function(cus_id){
    const customer = dataBase.customer.filter(function(cus){
      return cus.id === cus_id;
    })[0];
    return new Promise(function(resolve, reject) {
      setTimeout(function () {
        if(customer){
          resolve(customer);
        } else {
          reject("NO_CUSTOMER_FOUND");
        }
      }, this.loadTime);
    });
  }
}
```

load time: 1000ms

```
function getCustomerById(id) {
  return Promise.race([server1, server2, server3].map(function(server){
    return server.getCustomerById(id);
  }));
}
```

Fastest wins!

```
getCustomerById(3)
  .then(function(customer){
    console.log("got customer: " + JSON.stringify(customer));
  })
```

`Promise.race(iterable)`

- **arguments**: `Array<Promises>`
- **return**: `new Promise()` that resolves when **one** of the promises in the argument array have resolved

- **.then** callback receives the resolved value from the one promise that resolved.

Quiz!!!

What is the difference between these 4?

1

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

2

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

3

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

4

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

Quiz answer

1

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

2

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

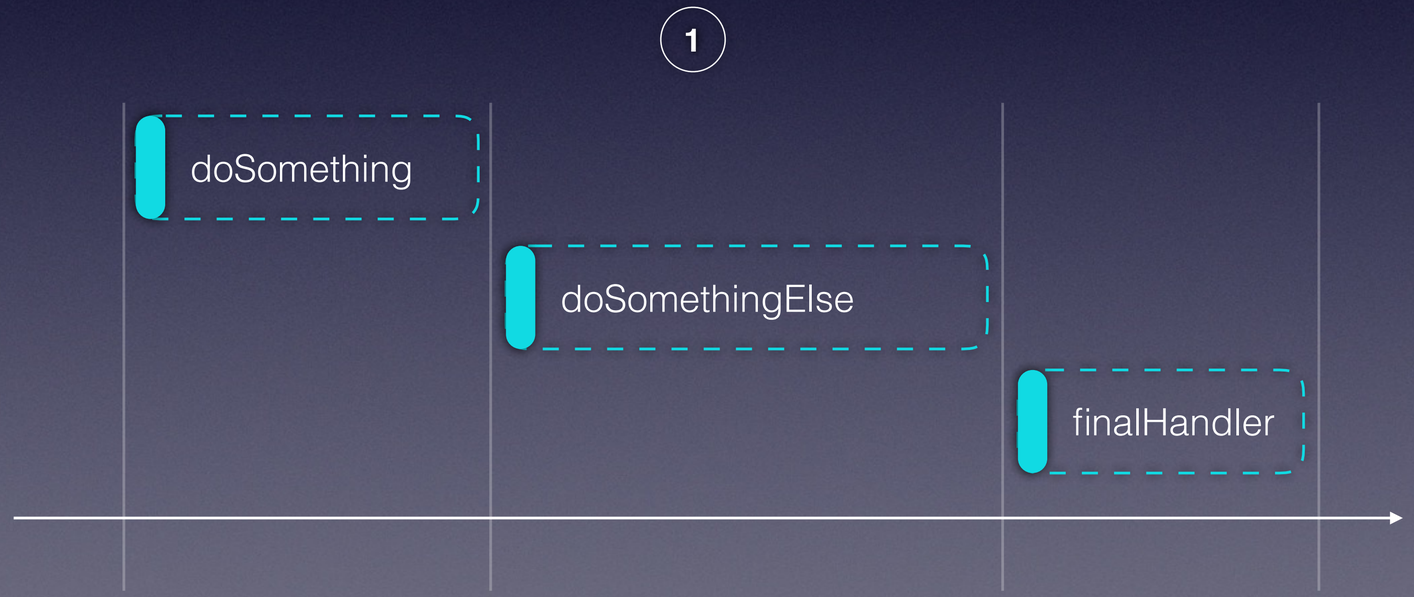
3

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

4

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

1



Quiz answer

1

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

2

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

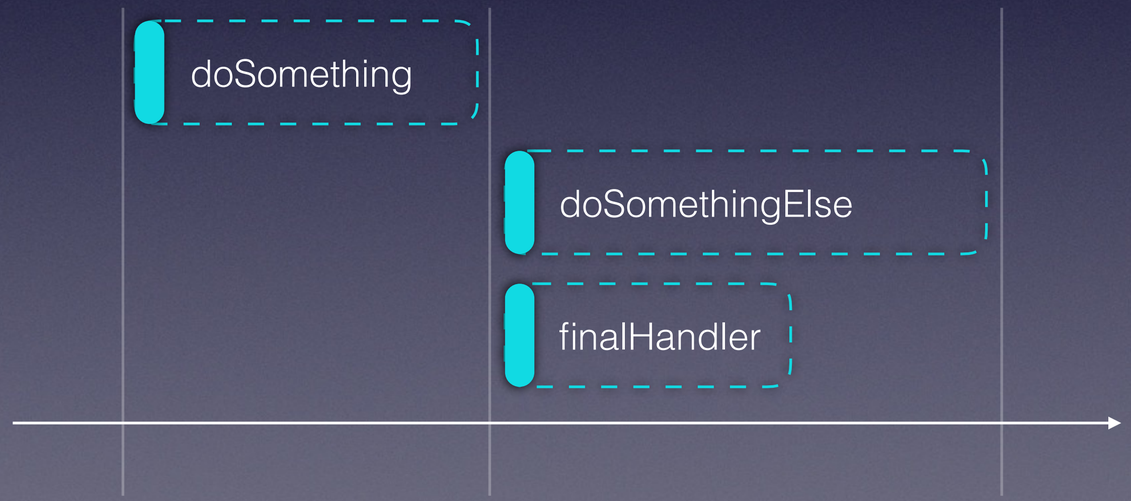
3

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

4

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

2



Quiz answer

1

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

2

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

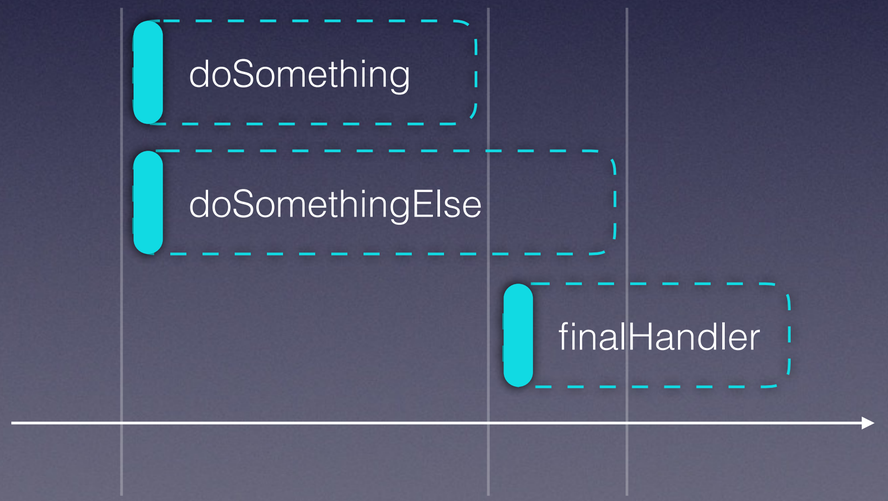
3

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

4

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

3



Quiz answer

1

```
doSomething()
  .then(function () {
    return doSomethingElse();
  })
  .then(finalHandler);
```

2

```
doSomething()
  .then(function () {
    doSomethingElse();
  })
  .then(finalHandler);
```

3

```
doSomething()
  .then(doSomethingElse())
  .then(finalHandler);
```

4

```
doSomething()
  .then(doSomethingElse)
  .then(finalHandler);
```

4