

# Software Studio

## 軟體設計與實驗

# Asynchronous

**Hung-Kuo Chu**

Department of Computer Science  
National Tsing Hua University

**CS2410**



# Asynchronous

- Synchronous vs. Asynchronous
  - Synchronous codes are executed **line by line**.
  - Asynchronous codes don't have to wait for the previous codes.
  - Asynchronous just means 'takes some time' or 'happens in the future, not right now'.
- Note that it doesn't mean it's multi-threaded, JavaScript can have asynchronous code, but it is generally **single-threaded**.



# Asynchronous (Cont'd)

- We use **asynchronous program** to listen to events and then execute functions.
- After the event is triggered, some code will be executed, it's called **event handler**.
- In the following case, “click” is the event fired, “console.log()” is the event handler.

```
var button = document.getElementById('myButton')
button.addEventListener( 'click', function(){
  console.log('hello!')})
```



# Asynchronous: Example

- The **setTimeout** function is a typical way that JS executes codes asynchronously.

```
console.log("Hello.");

setTimeout(function() {
  console.log("Goodbye!"); // Say "Goodbye" after two seconds from now.
}, 2000);

console.log("Hello again!");
// But setTimeout does not pause the execution of the code. It only
// schedules something to happen in the future, and then immediately
// continues to the next line.
```

```
Hello.           index.js:1
Hello again!     index.js:7
Goodbye!         index.js:4
```



# Asynchronous: Problem

```
var img1 = downloadPhoto('http://coolcats.com/cat.gif');  
// downloadPhoto is an aync function and takes some time to finish...  
img1.addEventListener('click', function() {});  
// img1 is 'undefined'!
```

- In this example, if the image **img1** is not loaded before execute **addEventListener**, an error will occur.
- If you have a lot of images to be loaded in html, it will cause trouble.
- Thus, we need to handle the download process (or **img1**) **asynchronously**.



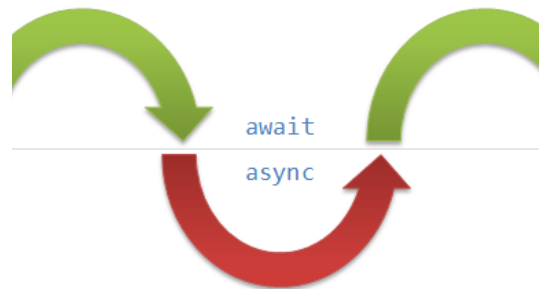
# Three Approaches



Callback



Promise



chron

Async / Await





Asynchronous

**CALLBACK**



# Callback Function

- We want to make sure the image is completely loaded before using it.
- We need a function to notify us whether the image loading is succeeded or failed.
  - **Callback function** (call me back when you're done)

```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto);  
// downloadPhoto is an aync function and takes some time to finish...  
  
// This function handles the result of downloadPhoto asynchronously.  
function handlePhoto (error, photo) {  
  if (error) console.error('Download error!', error)  
  else console.log('Download finished', photo)  
}
```



# Callback Function (Cont'd)

- Note that the **handlePhoto** is not invoked yet, it is just created and passed as a callback into **downloadPhoto**.
- It won't run until **downloadPhoto** finishes doing its task, which could take a long time depending on how fast the Internet connection is.



# Callback Function (Cont'd)

- Instead of immediately returning some result like most functions, functions that use callbacks take some time to produce a result, e.g., downloading things, reading files, talking to databases, etc.
- Basically, callback function is **using a function as the parameter of another function** and called by another function.



# Callback Example

```
function doHomework(subject, callback) {  
    alert(`Starting my ${subject} homework.`);  
    callback();  
}
```

// The callback function

```
function alertFinished(){  
    alert('Finished my homework');  
}  
doHomework('math', alertFinished);
```



# Callback Example (Cont'd)

```
function doHomework(subject, callback) {  
  alert(`Starting my ${subject} homework.`);  
  callback();  
}
```

// You can also write the callback function in anonymous function style

```
doHomework('math', function() {  
  alert('Finished my homework');  
});
```



# Callback Hell

- Sometimes we have a series of tasks where each step depends on the results of the previous step.
- This is a very straightforward thing to deal with in synchronous code:

```
var text = readFile(fileName),  
tokens = tokenize(text),  
parseTree = parse(tokens),  
optimizedTree = optimize(parseTree),  
output = evaluate(optimizedTree);  
console.log(output);
```


# Callback Hell (Cont'd)

- When you try to do this in asynchronous codes, it easily runs into **callback hell**.
- Callback functions are deeply nested inside of each other.

```
readFile(fileName, function(text) {  
  tokenize(text, function(tokens) {  
    parse(tokens, function(parseTree) {  
      optimize(parseTree, function(optimizedTree) {  
        evaluate(optimizedTree, function(output) {  
          console.log(output);  
        });  
      });  
    });  
  });  
});  
});
```

# Callback Hell (Cont'd)

```
1 function hell(win) {  
2   // for listener purpose  
3   return function() {  
4     loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {  
5       loadLink(win, REMOTE_SRC+'/lib/async.js', function() {  
6         loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {  
7           loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {  
8             loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {  
9               loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {  
10                loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {  
11                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {  
12                  loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {  
13                   async.eachSeries(SRIPTS, function(src, callback) {  
14                    loadScript(win, BASE_URL+src, callback);  
15                   });  
16                  });  
17                 });  
18                });  
19               });  
20              });  
21             });  
22            });  
23           });  
24          });  
25         });  
26        }  
}
```



# Callback Hell (Cont'd)

- Make your codes difficult to read and maintain.
- One of the solution is splitting the code into different functions with appropriate names (make it flat).





# Flat Callback Structure

```
function readFinish(text) {  
  tokenize(text, tokenizeFinish);  
}  
function tokenizeFinish(tokens) {  
  parse(tokens, parseFinish);  
}  
function parseFinish(parseTree) {  
  optimize(parseTree, optimizeFinish);  
}  
function optimizeFinish(optimizedTree) {  
  evalutate(optimizedTree, evaluateFinish);  
}  
function evaluateFinish(output) {  
  console.log(output);  
}  
readFile(fileName, readFinish);
```

The slide features several stylized purple flowers of varying sizes scattered across the background. One large flower is in the top right corner, another is in the bottom left, and several smaller ones are in the middle and bottom right.

Asynchronous

**PROMISE**



# Promise

- Instead of using functions that accept inputs and a callback, we make a function that returns a **promise** object.
- Promise is an object representing the execution status (**success** or **failure**) of an **asynchronous** operation.
  - in effect, a promise that a result of some kind will be returned at some point in the future.
- Promises are supported in ES6 or later.



# Promise (Cont'd)

- Promise is the browser's way of saying "I promise to get back to you with the answer as soon as I can", and it returns only two status: **succeed** or **fail**.
- A promise can only **succeed** or **fail once**. It cannot succeed or fail twice, and it cannot switch from success to failure or vice versa once the operation has completed.



# Promise (Cont'd)

- To use Promise, we have to new a Promise object with two parameters included in the function constructor: **resolve** (succeed) and **reject** (fail).



# Promise (Cont'd)

- Resolve code will be executed when the process is succeeded, or the return value is legal.

```
function asyncFunction(value) {  
  return new Promise(function(resolve, reject){  
    // ... do something asynchronous here ...  
    if(value){  
      resolve("Stuff worked!"); // succeed!  
    }else{  
      reject(Error("It broke")); // error 、 already rejected 、 failed  
    }  
  });  
}
```

# Promise (Cont'd)

- Both of `resolve` and `reject` have a return value, we can use `.then()/.catch()` to pass this value to next process.
- The `then()` method includes two parameters: **successCallback** and **failureCallback**, `failureCallback` is optional, kind of the `try/catch`.
- The `catch()` method handles error message.



# Example using Callback

```
function successCallback(result) { console.log("Audio file ready at URL: " + result); }  
  
function failureCallback(error) { console.log("Error generating audio file: " + error); }  
  
function doSomething (successCBF, failureCBF) {  
    // ...do some serious tasks here...  
    if (success) successCBF();  
    else failureCBF();  
}  
  
// usage  
doSomething(successCallback, failureCallback);
```





# Example using Promise

```
function successCallback(result) { console.log("Audio file ready at URL: " + result); }

function failureCallback(error) { console.log("Error generating audio file: " + error); }

// No callbacks are passed to the main function!
function doSomething () {
    return new Promise(function(resolve, reject){
        // ...do some serious tasks here...
        if(success){
            resolve("Stuff worked!") // succeed!
        }else{
            reject(Error("It broke")) // error 、 already rejected 、 failed
        }
    });
}

// usage
const promise = doSomething();
promise.then(successCallback, failureCallback);
```



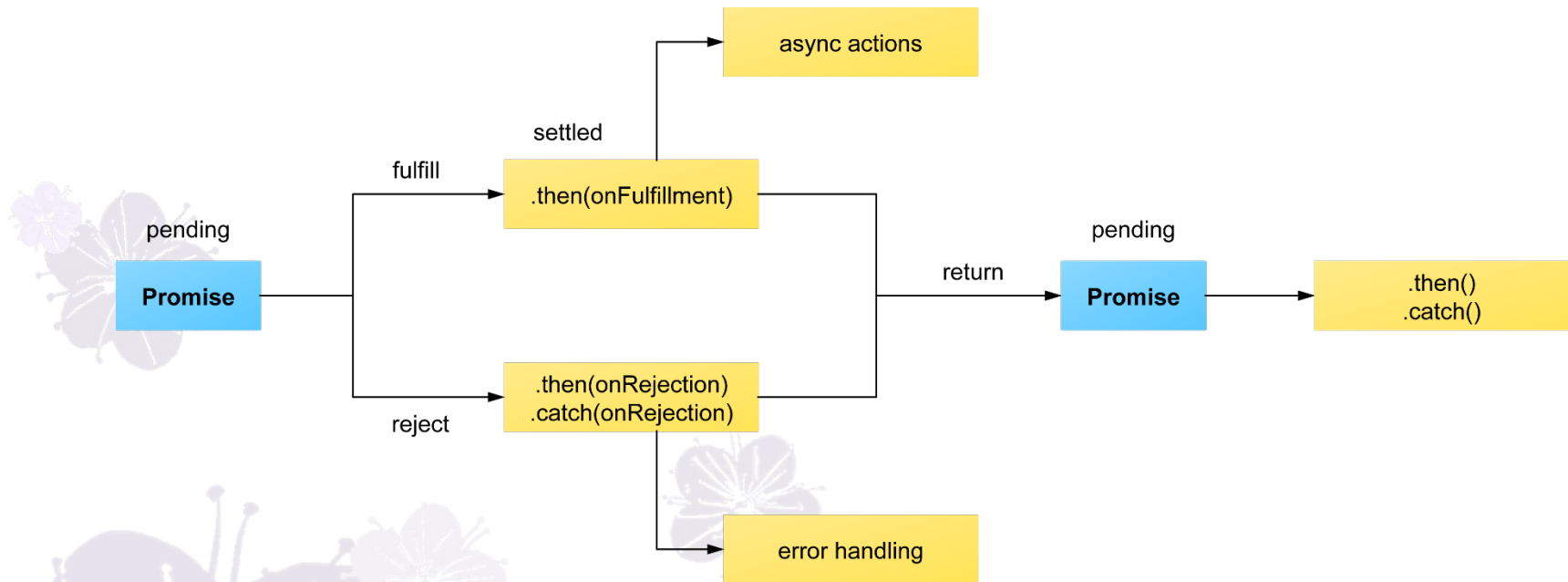
# Promise Terminology

- When a promise is created, it is neither in a success or failure state. It is said to be **pending**.
- When a promise returns, it is said to be **resolved**.
- A successfully resolved promise is said to be **fulfilled**.
  - It returns a value, which can be accessed by chaining a **.then()** block onto the end of the promise chain.
- An unsuccessfully resolved promise is said to be **rejected**.
  - It returns an error message stating why the promise was rejected, which can be accessed by chaining a **.catch()** block onto the end of the promise chain.



# Promise Concept

```
new Promise( /* executor */ function(resolve, reject) { ... } );
```



# Syntactic Sugar – Arrow Function

```
function(a, b, c) {  
  return doSomethingElse(a, b, c);  
}
```



```
(a, b, c) => {return doSomethingElse(a, b, c);}
```



```
(a, b, c) => doSomethingElse(a, b, c)
```

If there is only one argument / parameter

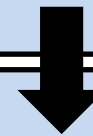
```
a => doSomethingElse(a)
```



# Promise - Constructor

```
function asyncFunc () {  
  return new Promise(function(resolve, reject){  
    // do some asynchronous tasks here...  
    // depends on the outcome to call either  
    resolve(someValue); // succeed!  
    // or  
    reject("failure reason"); // rejected!  
  });  
}
```


Equals to...



```
let asyncFunc = new Promise((resolve, reject) => {  
  // do some asynchronous tasks here...  
  // depends on the outcome to call either  
  resolve(someValue); // succeed!  
  // or  
  reject("failure reason"); // rejected!  
});
```

# Promise - Constructor

```
let myFirstPromise = new Promise((resolve, reject) => {  
  // In this example, we use setTimeout(...) to simulate async code.  
  // In reality, you will probably use something like XHR or an HTML5 API.  
  setTimeout( function() {  
    resolve('Success!');  
    // or  
    // reject ("Error!");  
  }, 500);  
});  
  
myFirstPromise.then(successMessage => {  
  // successMessage is whatever we passed in the resolve(...) function above.  
  console.log("Yay! " + successMessage);  
}, (errorMessage) => {  
  // errorMessage is whatever we passed in the reject(...) function above.  
  console.log("No! " + errorMessage);  
});
```



# .then()

- The then() method returns a **Promise**. It takes up to two arguments: callback functions for the **success** and **failure** cases of the Promise.

```
p.then(onFulfilled[, onRejected]);
```

```
p.then(function(value) {  
  // fulfillment  
}, function(errorMessage) {  
  // handle the rejection  
});
```

```
p.then(onFulfilled[, onRejected]);
```

```
p.then((value) => {  
  // fulfillment  
}, (errorMessage) => {  
  // handle the rejection  
});
```



# .then() (Cont'd)

- Once a **Promise** is fulfilled or rejected, the respective handler function (**onFulfilled** or **onRejected**) will be called asynchronously (scheduled in the current thread loop).
- The behavior of the handler function follows a specific set of rules.

```
let p = new Promise((resolve, reject) => {  
  resolve();  
});
```

- Returns a value:** the promise returned by then() will be **resolved** with the returned value as its value.

```
let p2 = p.then( () => {  
  return value;  
});
```



```
let p2 = new Promise((resolve, reject) => {  
  resolve(value);  
});
```





# .then() (Cont'd)

2. **Returns nothing**: the promise returned by then() gets **resolved** with an undefined value.

```
let p2 = p.then( () => {  
  // return;  
});
```



```
let p2 = new Promise((resolve, reject) => {  
  resolve();  
});
```

3. **Throws an error**: the promise returned by then() gets **rejected** with the thrown error as its value.

```
let p2 = p.then( () => {  
  throw value;  
});
```



```
let p2 = new Promise((resolve, reject) => {  
  reject(value);  
});
```



# .then() (Cont'd)

4. **Returns an already fulfilled promise:** the promise returned by then() gets **fulfilled** with that promise's value as its value.

```
let p2 = p.then( () => {  
  return Promise.resolve(value);  
});
```



```
let p2 = new Promise((resolve, reject) => {  
  resolve(value);  
});
```

5. **Returns an already rejected promise:** the promise returned by then() gets rejected with that promise's value as its value.

```
let p2 = p.then( () => {  
  return Promise.reject(value);  
});
```




```
let p2 = new Promise((resolve, reject) => {  
  reject(value);  
});
```



# .then() (Cont'd)

**6. Returns another pending promise object:** the resolution/rejection of the promise returned by then() will be subsequent to the resolution/rejection of the promise returned by the handler. Also, the resolved value of the promise returned by then() will be the same as the resolved value of the promise returned by the handler.

```
let p2 = p.then( () => {  
  return new Promise((resolve, reject) => {  
    resolve(value);  
    // or  
    // reject(value);  
  });  
});
```



```
let p2 = new Promise((resolve, reject) => {  
  resolve(value);  
  // or  
  // reject(value);  
});
```



# .catch()

- The catch() method returns a **Promise** and deals with **rejected cases only**. It behaves the same as calling **then(undefined, onRejected)**

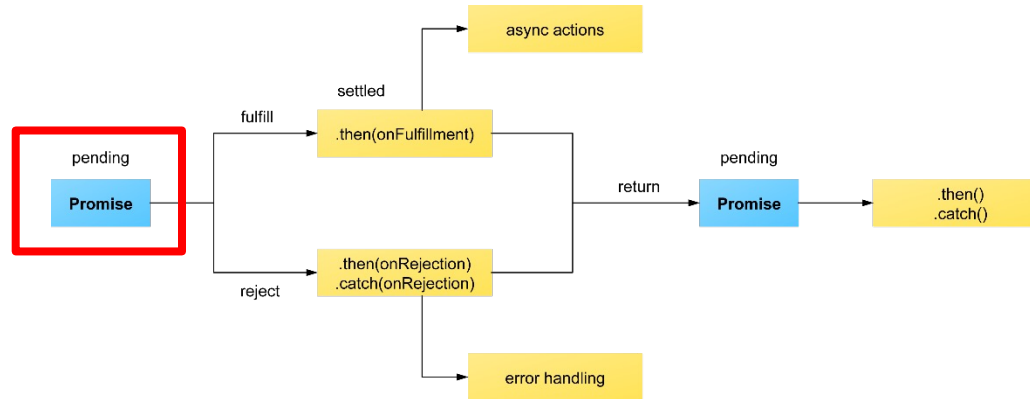
```
p.catch(function(reason) {  
  // handle the rejection  
});
```



```
p.then(undefined, function(reason) {  
  // handle the rejection  
});
```



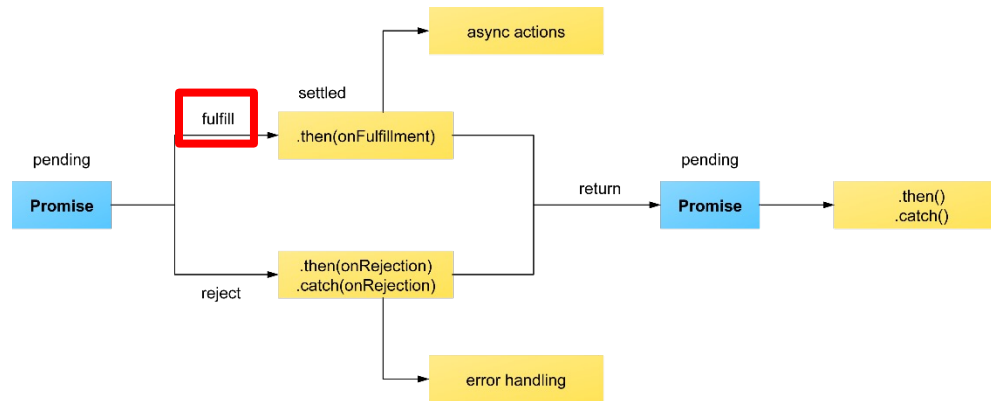
# Chaining



```
let p = new Promise(function(resolve, reject) {  
  resolve(1);  
});
```

```
p.then(function(value) {  
  console.log(value); // 1  
  return value + 1;  
}).then(function(value) {  
  console.log(value + '- This synchronous usage is virtually pointless');  
  // 2- This synchronous usage is virtually pointless  
});
```

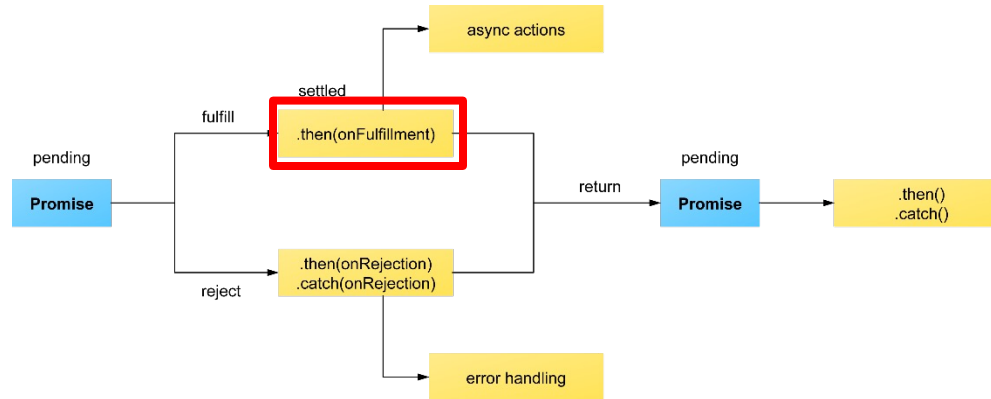
# Chaining



```
let p = new Promise(function(resolve, reject) {  
  resolve(1);  
});
```

```
p.then(function(value) {  
  console.log(value); // 1  
  return value + 1;  
}).then(function(value) {  
  console.log(value + '- This synchronous usage is virtually pointless');  
  // 2- This synchronous usage is virtually pointless  
});
```

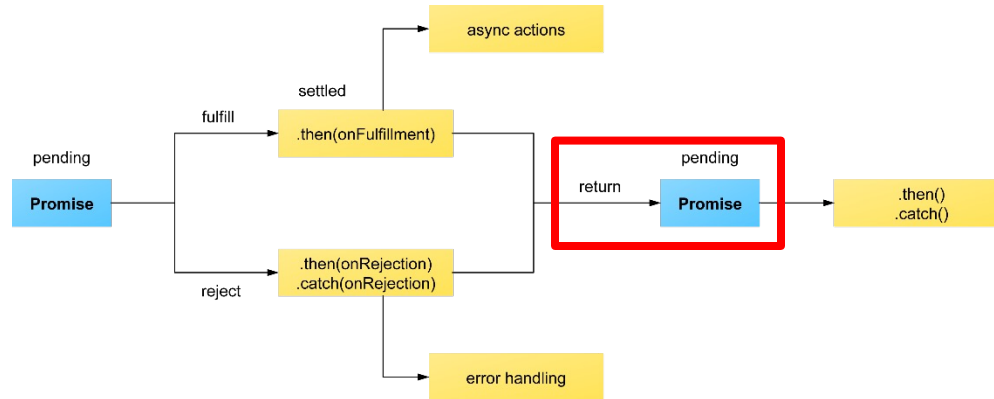
# Chaining



```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

p.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + '- This synchronous usage is virtually pointless');
  // 2- This synchronous usage is virtually pointless
});
```

# Chaining

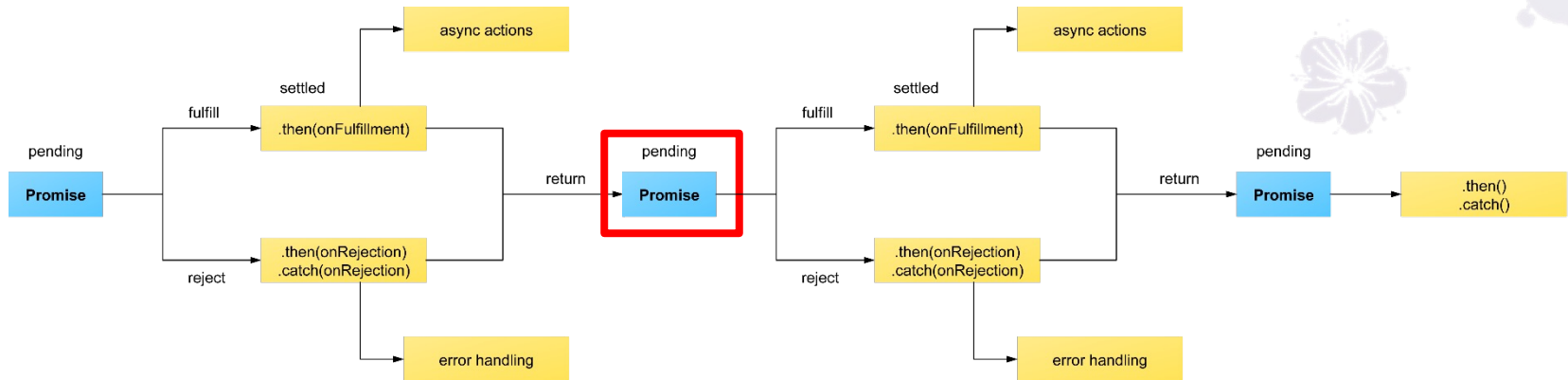


```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

p.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + '- This synchronous usage is virtually pointless');
  // 2- This synchronous usage is virtually pointless
});
```



# Chaining



```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});
```

**Or you can write codes like these...**

```
p.then(function(value) {
  console.log(value); // 1
```

```
  return value + 1;
```

```
}).then(function(value) {
```

```
  console.log(value + '- This synchronous usage is virtually pointless');
```

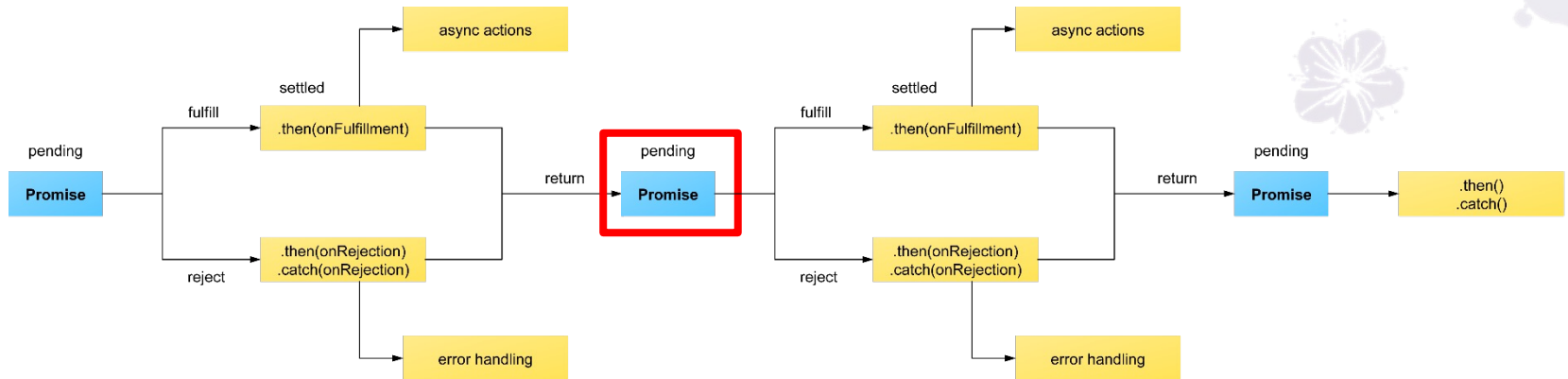
```
  // 2- This synchronous usage is virtually pointless
```

```
});
```

```
return new Promise(function(resolve, reject) {
  resolve(value + 1);
});
```

```
return Promise.resolve(value + 1);
```

# Chaining



```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});
```

Or you can write codes like these...

```
p.then(function(value) {
  console.log(value); // 1

```

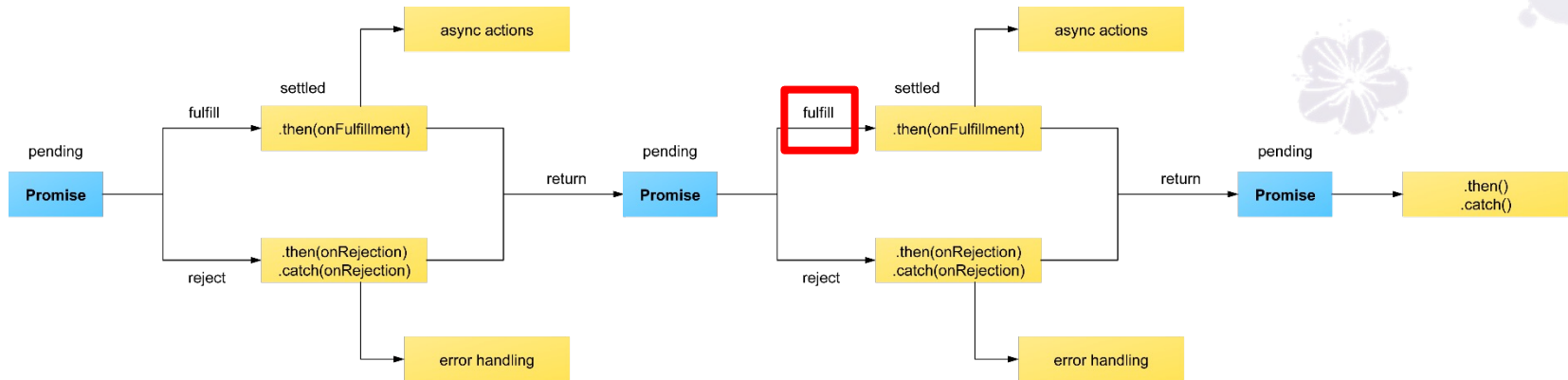
```
  return value + 1;
}).then(function(value) {
```

```
return new Promise(function(resolve, reject) {
  resolve(value + 1);
});
```

```
return Promise.resolve(value + 1);
```

```
  console.log(value + '- This synchronous usage is virtually pointless');
  // 2- This synchronous usage is virtually pointless
});
```

# Chaining



```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});
```

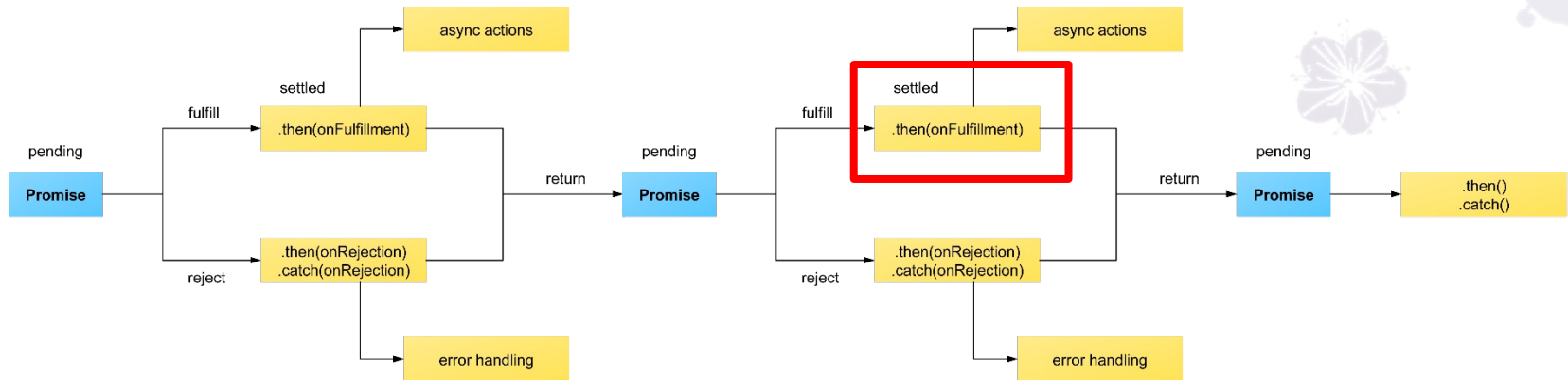
```
p.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + '- This synchronous usage is virtually pointless');
  // 2- This synchronous usage is virtually pointless
});
```

Or you can write codes like these...

```
return new Promise(function(resolve, reject) {
  resolve(value + 1);
});
```

```
return Promise.resolve(value + 1);
```

# Chaining



```
let p = new Promise(function(resolve, reject) {  
  resolve(1);  
});  
  
p.then(function(value) {  
  console.log(value); // 1  
  return value + 1;  
}).then(function(value) {  
  console.log(value + '- This synchronous usage is virtually pointless');  
  // 2- This synchronous usage is virtually pointless  
});
```

# Error Propagation

- If there's an exception, the browser will look down the chain for a nearest **.catch()** handlers

```
doSomething()  
.then(result => doSomethingElse(result))  
.then(newResult => doThirdThing(newResult))  
.then(finalResult => console.log('final result: ${finalResult};'))  
.catch(failureCallback);
```



# Error Propagation

- If there's an exception, the browser will look down the chain for a nearest **.catch()** handlers

If this function failed, it jumps directly to the .catch() handler without executing the next two lines

```
doSomething()  
.then(result => doSomethingElse(result))  
.then(newResult => doThirdThing(newResult))  
.then(finalResult => console.log('final result: ${finalResult};'))  
.catch(failureCallback);
```

These two lines will not be executed



# Error Propagation

- If there's an exception, the browser will look down the chain for a nearest **.catch()** handlers

If this function failed, it jumps directly to the **.catch()** handler without executing the next two lines

```
doSomething()  
.then(result => doSomethingElse(result))  
.then(newResult => doThirdThing(newResult))  
.then(finalResult => console.log('final result: ${finalResult};'))  
.catch(failureCallback)  
.then(anotherResult => doOtherThing(anotherResult);
```

After handling the exception, the browser will continue executing **.then()** chained after **.catch()**.



# Promise vs. Callback

- Supports chaining
  - Chains multiple async operations together using **multiple .then()** operations, passing the result of one into the next one as an input.
  - Using callbacks leads to callback hell!
- Strict execution order
  - Promise callbacks are always called in the strict order they are placed in the event queue.
- Better error handling
  - All errors are handled by a single .catch() block at the end of the block, rather than being individually handled in each level of the "pyramid".





# Let's Order a Pizza!

1. You choose what toppings you want.
  - This can take a while if you are indecisive and may fail if you just can't make up your mind or decide to get a curry instead.
2. You then place your order.
  - This can take a while to return a pizza and may fail if the restaurant does not have the required ingredients to cook it.
3. You then collect your pizza and eat.
  - This might fail if, say, you forgot your wallet so can't pay for the pizza!



# Callback Version

```
chooseToppings(function(toppings) {  
  placeOrder(toppings, function(order) {  
    collectOrder(order, function(pizza) {  
      eatPizza(pizza);  
    }, failureCallback);  
  }, failureCallback);  
}, failureCallback);
```

- Code is hard to read: Callback hell.
- failureCallback() are called multiple times.



# Promise Version

```
chooseToppings()  
  .then(function(toppings) {  
    return placeOrder(toppings);  
  })  
  .then(function(order) {  
    return collectOrder(order);  
  })  
  .then(function(pizza) {  
    eatPizza(pizza);  
  })  
  .catch(failureCallback);
```

```
chooseToppings()  
  .then(toppings =>  
    placeOrder(toppings)  
  )  
  .then(order =>  
    collectOrder(order)  
  )  
  .then(pizza =>  
    eatPizza(pizza)  
  )  
  .catch(failureCallback);
```

```
chooseToppings().then(placeOrder).then(collectOrder).then(eatPizza)  
  .catch(failureCallback);
```



# Promise - all

- Execute multiple promises at once
- If all the promises succeed:
  - Return an array of multiple resolved values
- One of the promises failed:
  - Return one rejected value



# Promise - all



```
var p1 = Promise.resolve(3);  
var p2 = 1337;  
var p3 = new Promise((resolve, reject) => { setTimeout(resolve, 100, 'foo'); });  
  
Promise.all([p1, p2, p3]).then(values => { console.log(values); }); // [3, 1337, "foo"]
```

```
var p1 = Promise.resolve(3);  
var p2 = 1337;  
var p3 = new Promise((resolve, reject) => { setTimeout(reject, 100, 'foo'); });  
  
Promise.all([p1, p2, p3]).then(values => { console.log(values); }) // print nothing  
.catch(errMessage => { console.log(errMessage); }); // print 'foo'
```



# Promise - race

- Execute multiple promises at once
- Return any value that **firstly** gets either **resolved** or **rejected**.



# Promise – race



```
var p1 = Promise.resolve(3);  
var p2 = new Promise((resolve, reject) => { setTimeout(resolve, 100, 'foo'); });  
  
Promise.race([p1, p2]).then(values => { console.log(values); }); // 3
```

```
var p1 = new Promise((resolve, reject) => { setTimeout(resolve, 100, 'foo'); });  
var p2 = new Promise((resolve, reject) => { setTimeout(reject, 10, 'failed'); });  
  
Promise.race([p1, p2]).then(values => { console.log(values); }) // print nothing  
.catch(errMessage => { console.log(errMessage); }); // print 'failed'
```





Asynchronous

# **ASYNC / AWAIT**





# The **async** Keyword

- Using the **async** keyword to turn a function into an asynchronous function.
- An async function knows to invoke the asynchronous code with the **await** keyword.
- An async function **ALWAYS** returns a promise.

```
async function hello() { return "Hello" };
```

```
hello(); // since it returns a promise, we can use .then() as follows..
```

```
hello().then((value) => console.log(value));
```

# The **await** Keyword

- The **await** keyword **ONLY** works **inside** async functions.
- Putting the **await** keyword in front of any async promise-based function will pause the code until the promise fulfills/rejects.
- **await** affects the execution order of functions within an async function.



# Async / Await: Example

This is a promise that will be resolved after 2s

```
async function asyncRun() {  
  let jamesRun = await runPromise('James', 2000);  
  console.log('Finished: ', jamesRun);  
  let claireRun = await runPromise('Claire', 2500);  
  console.log('Finished: ', claireRun);  
}
```

This is a promise that will be resolved after 2.5s

- The outputs will be:

Finished: James -> Finished: Claire



# Async / Await Example

```
function resolveAfter2Seconds(x) {  
  return new Promise(resolve => {  
    setTimeout(() => { resolve(x); }, 2000);  
  });  
}
```

```
async function add1(x) {  
  const a = await resolveAfter2Seconds(20);  
  const b = await resolveAfter2Seconds(30);  
  return x + a + b;  
}
```

```
add1(10).then(v => { console.log(v); }); // prints 60 after 4 seconds.
```



# Async / Await vs. Promise

```
fetch('coffee.jpg')
.then(response => {
  if (!response.ok) {
    throw new Error(`HTTP error! status:
    ${response.status}`);
  }
  return response.blob();
})
.then(myBlob => {
  let objectURL =
  URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
})
.catch(e => { console.log('There has been a
problem with your fetch operation: ' +
e.message);});
```

```
async function myFetch() {
  let response = await fetch('coffee.jpg');

  if (!response.ok) {
    throw new Error(`HTTP error! status:
    ${response.status}`);
  }

  let myBlob = await response.blob();

  let objectURL =
  URL.createObjectURL(myBlob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);
}

myFetch()
.catch(e => { console.log('There has been a
problem with your fetch operation: ' +
e.message); });
```

# Async / Await

- It seems that JavaScript can work fine without async/await
- Just promise can do many things
- Advantages of Async / Await:
  - More readable
  - More clean
  - Do more complex promise operation



# References

- [Learn Web Development: Asynchronous JavaScript](#)
- [鐵人賽：使用 Promise 處理非同步](#)
- [鐵人賽：JavaScript Await 與 Async](#)



thank  
you!

Question

