Software Studio
軟體設計與實驗

JavaScript – Part II

Hung-Kuo Chu
Department of Computer Science
National Tsing Hua University

CS2410

Codeblock Conventions

HTML5 Program

JavaScript Program

Review

- Variables
  - keywords: var, let, const
  - types: string, number, boolean, undifined
- Control Flow
  - if…else, for loop, while loop, switch, try catch
- Object and Array
  - Initialization, property, operations
- Function
  - anonymous/arrow/nested function, closures

Outline

- Document Object Model (DOM)
- jQuery
- Asynchronous

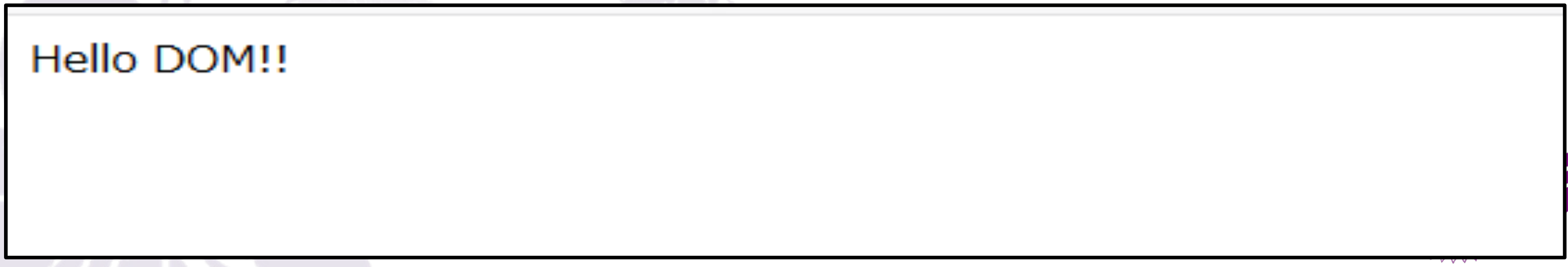
What is DOM?

Document **O**bject **M**odel

The whole HTML document is an object.
We can use JavaScript to control the HTML document.

A DOM Example

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>DOM example #1</title>
    <script type="text/javascript">
        function init() {
            var text = document.getElementById("dom1");
            text.innerHTML = "Hello DOM!!";
        }
    </script>
</head>
<body onload="init();">
    <p id="dom1"></p>
</body>
</html>
```



```
Hello DOM!!
```

DOM Example (Explained)

```
<html>
<head>
    <meta content="text/html; charset=UTF-
8">
    <title>DOM example #1</title>
    <script type="text/javascript">
     function init() { var text =
           document.getElementById("dom1");
           text.innerHTML = "Hello DOM!!";
    }
    </script>
</head>
<body onload="init();">
    <p id="dom1"></p>
</body>
</html>
```

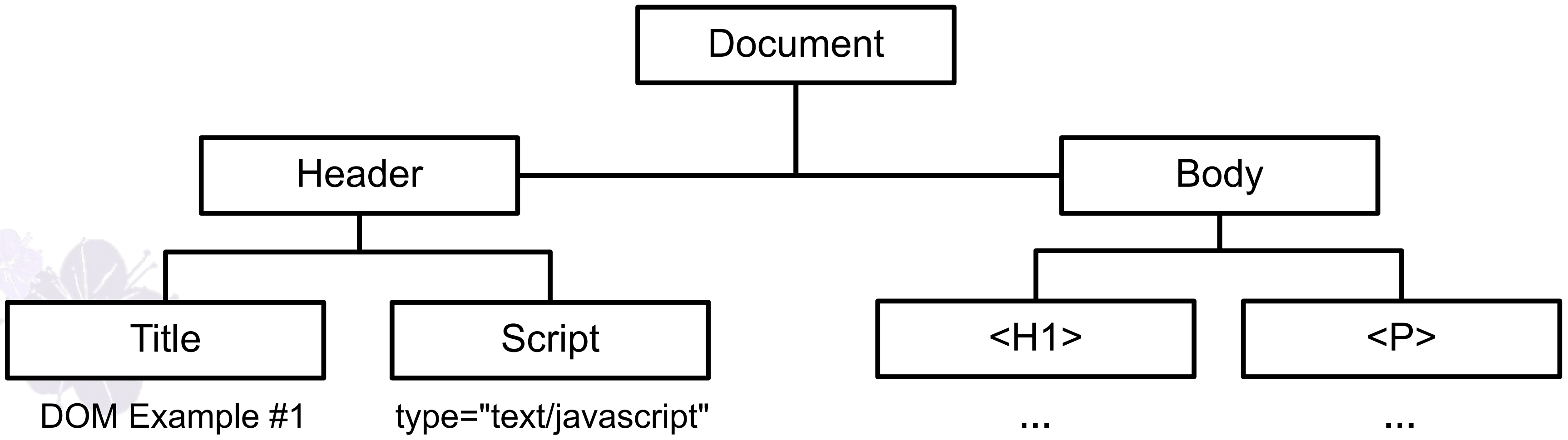
First, we use **getElementById** to get the object with specific id ("dom1“ in this example).

And then we use **innerHTML** to modify the **content** of this object to display our string.

The <p> is an object in JavaScript with "**dom1**“ as its id.

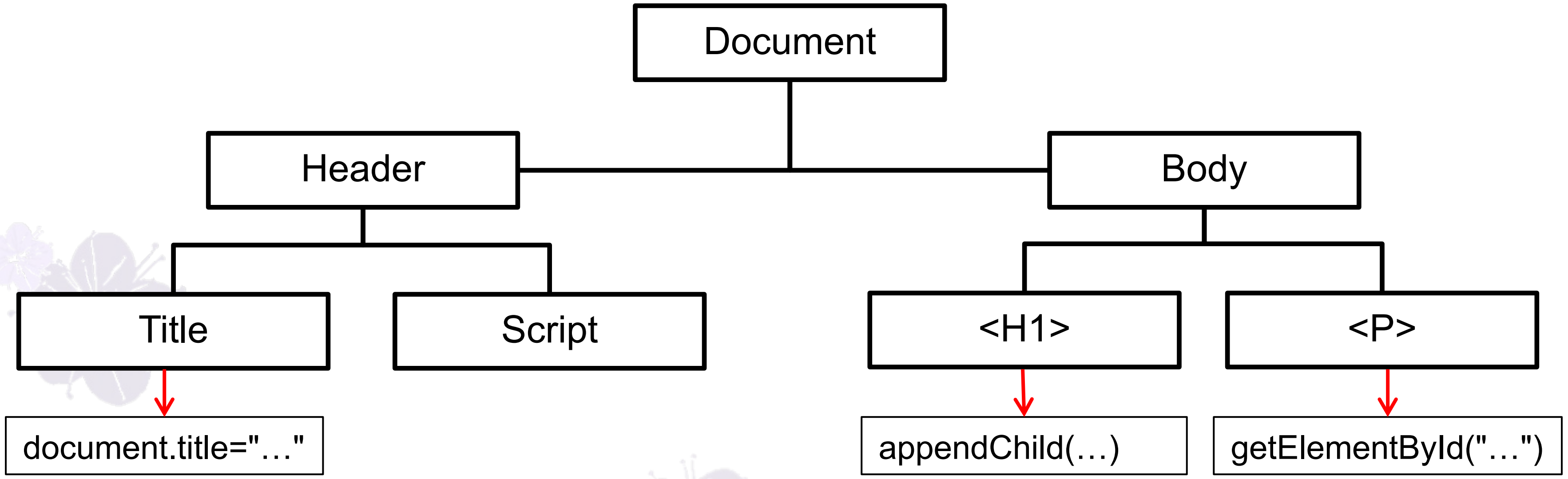
We can use getElementById to modify its content.

More about DOM



HTML page is a tree structure with many nodes.
Each node has its own data and attribute(s).

More about DOM



In JavaScript, we can do operations on these nodes. That’s how a dynamic web page works!

Finding Objects in DOM

- document.getElementById(id_name):
  - Find a node by its id.
- document.getElementsByClassName(class_name)
  - Find all nodes with the specified class name.
- document.getElementsByTagName (tag_name)
  - Find all nodes with the specified tag name.

```
<p class="class_name" id="id_name">Content</p>
```

Finding Objects in DOM (Cont’d)

- node.previousSibling
- node.nextSibling
- node.firstChild
- node.lastChild
- node.hasChildNodes()
- node.parentNode
- node.appendChild(new_node)
  - Insert a node.
- node.removeChild(old_node)
  - Remove a child node.
- node.replaceChild(old_node, new_node)
  - Replace a child node.
- node.cloneNode(true)
  - Clone a node and its element.
- document.createElement(tag)
  - Create a node with tag.

JavaScript Element and Node

Manipulating Objects in DOM

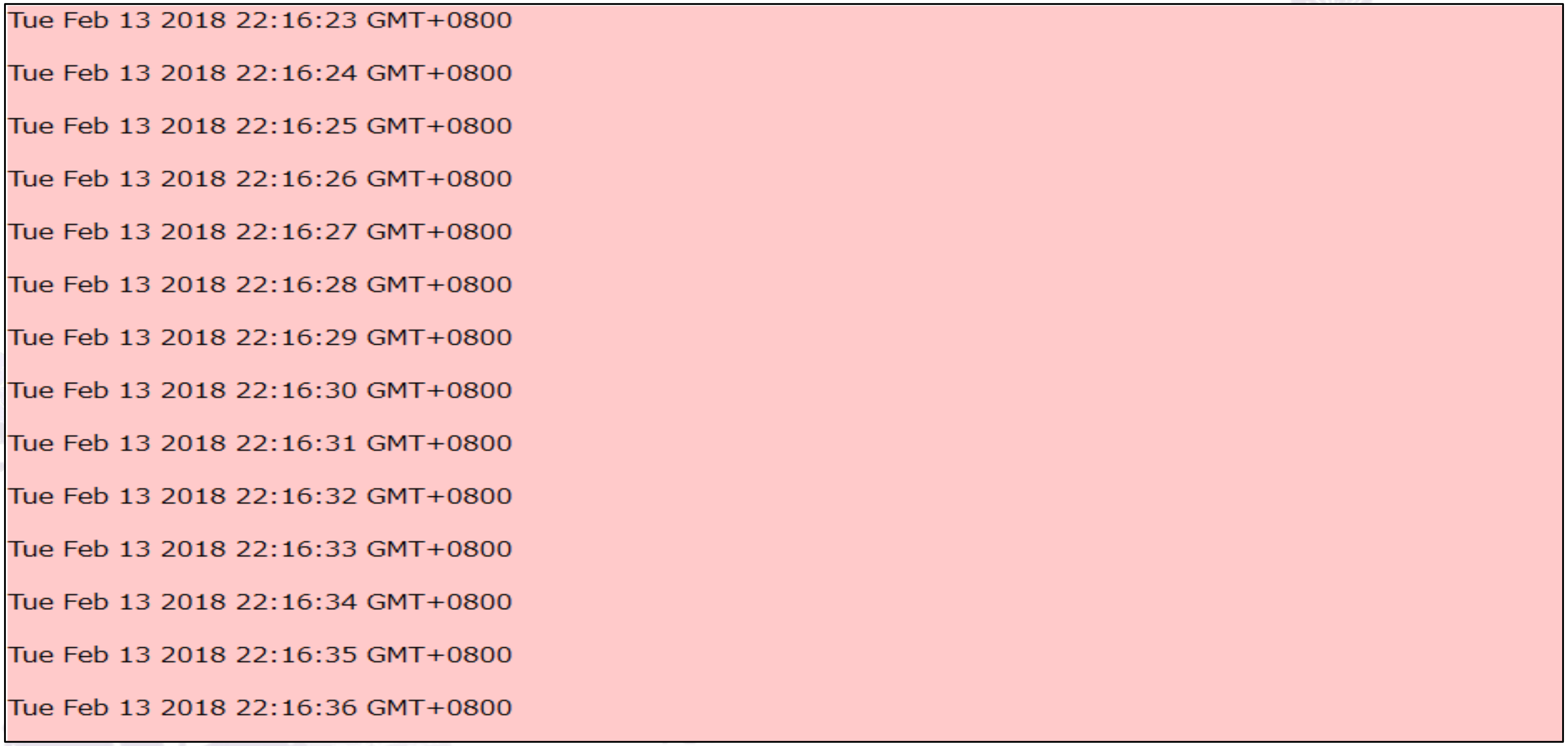
- node.style.css_attribute:
  - Get or set CSS attribute.
- node.innerHTML:
  - Get or set HTML content of a node.
- node.attribute_name:
  - Get or set HTML attribute of a node.

```
<button id="id" autofocus style="font-size:50pt">Click!!</button>
```

DOM Example

```
<html>
<head>
   <meta content="text/html; charset=UTF-8">
   <title>DOM example #2</title>
   <script type="text/javascript">
      var timer = setInterval(appendDate, 1000);

      function appendDate() {
          var sect = document.getElementById("container");
          var text = document.createElement("p");
          text.innerHTML = Date();
          sect.appendChild(text);
       }
   </script>
</head>
<body>
   <section id="container" style="background:#ffcaca"></section>
</body>
</html>
```

DOM Example: Result



```
Tue Feb 13 2018 22:16:23 GMT+0800

Tue Feb 13 2018 22:16:24 GMT+0800

Tue Feb 13 2018 22:16:25 GMT+0800

Tue Feb 13 2018 22:16:26 GMT+0800

Tue Feb 13 2018 22:16:27 GMT+0800

Tue Feb 13 2018 22:16:28 GMT+0800

Tue Feb 13 2018 22:16:29 GMT+0800

Tue Feb 13 2018 22:16:30 GMT+0800

Tue Feb 13 2018 22:16:31 GMT+0800

Tue Feb 13 2018 22:16:32 GMT+0800

Tue Feb 13 2018 22:16:33 GMT+0800

Tue Feb 13 2018 22:16:34 GMT+0800

Tue Feb 13 2018 22:16:35 GMT+0800

Tue Feb 13 2018 22:16:36 GMT+0800
```

DOM Example (Explained)

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>DOM example #2</title>
    <script type="text/javascript">
        var timer = setInterval(appendDate, 1000);

        function appendDate() {
            var sect = document.getElementById("container");
            var text = document.createElement("p");
            text.innerHTML = Date();
            sect.appendChild(text);
        }
    </script>
</head>
<body>
    <section id="container" style="background:#ffcaca"></section>
</body>
</html>
```

With ‘**setInterval**’, we can define some time event let JavaScript do something after a period. (In this case ‘appendDate’, executed in every 1000 ms)

DOM Example (Explained)

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>DOM example #2</title>
    <script type="text/javascript">
        var timer = setInterval(appendDate, 1000);

        function appendDate() {
            var sect = document.getElementById("container");
            var text = document.createElement("p");
            text.innerHTML = Date();
            sect.appendChild(text);
        }
    </script>
</head>
<body>
    <section id="container" style="background:#ffcaca"></section>
</body>
</html>
```

Use the getElementById to find message area object

DOM Example (Explained)

```
<html>
<head>
  <meta content="text/html; charset=UTF-8">
  <title>DOM example #2</title>
  <script type="text/javascript">
     var timer = setInterval(appendDate, 1000);

     function appendDate() {
        var sect = document.getElementById("container");
        var text = document.createElement("p");
        text.innerHTML = Date();
        sect.appendChild(text);
     }
  </script>
</head>
<body>
   <section id="container" style="background:#ffcaca"></section>
</body>
</html>
```

1. Create a <p> node by using document.createElement("p")
2. Get current time by using Date()
3. Append to content of our new node.
4. Append the node to message area

We Have Learned…

- ✓ HTML document is an object with tree structure.
- ✓ We can edit HTML document dynamically with JavaScript using DOM.

About jQuery

- “write less, do more” -- jQuery is a library that makes the writing of JavaScript faster.
- With jQuery, we can do complex things with few lines of codes.
- Some companies like Google also use jQuery to design their webpages.

About jQuery

- We can do the following things with jQuery:
  - HTML/DOM manipulation.
  - CSS style manipulation.
  - HTML event handling.
  - Effects and animation.
  - AJAX (Asynchronous JavaScript + XML)

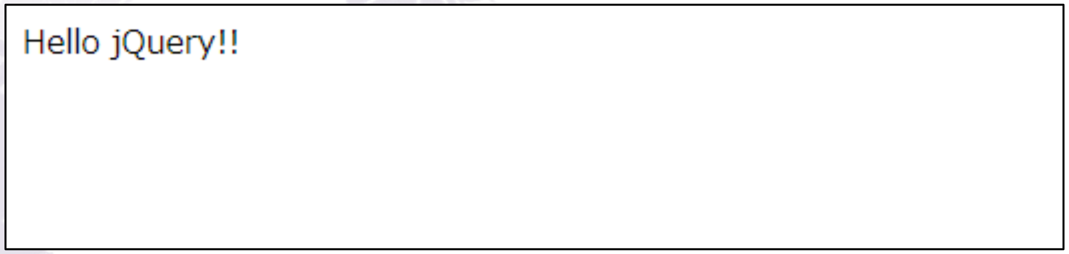
Using jQuery

- Usage #1: Download jQuery manually:
  - https://jquery.com/
  - <script src="jquery-3.6.0.min.js" type="text/javascript"></script>

- Usage #2: Linking jQuery from CDN:
  - <script src="http://code.jquery.com/jquery-3.6.0.js" type="text/javascript"></script>

jQuery Example

```
<html>
<head>
    <meta content="text/html; charset=UTF-8">
    <title>jQuery example #1</title>
    <script src="http://code.jquery.com/jquery-3.3.1.js" type="text/javascript"></script>
    <script>
        $(document).ready(function () {
            document.write("Hello jQuery!!");
        });
    </script>
</head>
<body>
</body>
</html>
```



```
Hello jQuery!!
```

jQuery Syntax

```
$(selector).action()
```

It means that we are using jQuery now.

Select an element

Do something with selected element.

```
$(selector).action1().action2().action3()
```

We can also make an action chain by appending multiple actions.

jQuery Selectors

- We can use selector to find element in HTML document.
  - $("#id_name"): Select a **single** element with **id** name.
  - $(".class_name"): Select **all** elements with **class** name.
  - $("tag_name"): Select **all** elements with **tag** name.

- We can also use multiple selectors to select element:
  - $("div.main"): Select elements with <div> tag and class 'main'.
  - $("h1, h2, h3"): Select all h1, h2 and h3 elements.

- More selectors
  - http://www.w3schools.com/jquery/jquery_selectors.asp

jQuery: Hide and Show

- We can use **hide**() or **show**() to hide or show an element.
  - $("p").hide(): Hide all elements with tag <p>.
  - $("p").show(): Show all elements with tag <p>.
- We can also use **toggle**() to switch between hide and show!
- Example

jQuery: Fade

- We can fade in/out an element by using **fadeIn**()/**fadeOut**():
  - $("#out").fadeOut(): Fade out element with id ‘out’.
  - $("#in").fadeIn(): Fade in element with id ‘in’.
  - $("#fadeto").fadeTo(): Turn transparency to a certain value.
- And we can set the length of animation:
  - $("p").fadeOut(5000): Fade out all <p> in 5 seconds.
- Example

jQuery: Animate

- We can add animation to an element by using **animate**().

Syntax: animate({*param*}, *speed*, *callback*)

Attribute we want to change

Length of animation(ms)

Do something after the animation.

Ex: $("div").animate({left: '250px'}, 1000)

Move <div> **right** 250 px in 1 second (The left attribute indicates the distance between element and left border).

- Example

jQuery: Add / Remove Elements

- Add Elements
  - **append**(*element*):
    - Inserts content at the end of the selected elements
  - **prepend**(*element*)
    - Inserts content at the beginning of the selected elements
- Remove Elements
  - **remove**()
    - Removes the selected element (and its child elements)
  - **empty**()
    - Removes the child elements from the selected element

jQuery: Element Contents

- Get/Set contents
  - **text**():
    - Gets (or sets) text content of an element.
  - **html**():
    - Gets (or sets) HTML content of an element.
  - **Val**():
    - Gets (or sets) the value of form fields
- Example: Get, Set.

jQuery: Element Attributes

- Get/Set attributes
  - Syntax: attr(*attribute_name*, *attribute*)
  - $("#img").attr("src")
    - Gets the src attribute of element with id ‘img’.
  - $("#img").attr("src", "SoftwareStudio.gif")
    - Changes the src attribute of element with id ‘img’.
  - You can set multiple attributes at once!
- Example: Get, Set.

jQuery: CSS

- Get/Set CSS attributes
  - Syntax: css(*attribute_name*, *attribute*).
  - $("#txt").css(“background-color”)
    - Get the background-color attribute of element with id ‘txt’.
  - $(“p”).css(“background-color”, “#FFFF00”)
    - Set the background color of all <p> elements to yellow.
  - You can set multiple attributes at once!
- Example: Get, Set.

jQuery: AJAX

- AJAX = **A**synchronous **J**avaScript **a**nd **X**ML.
- AJAX is the art of exchanging data with a server and updating parts of a web page without reloading the whole page.

jQuery: AJAX - Load

- The **load()** method loads data from a server and puts the returned data into the selected element.
- Syntax: load(*URL*, *data, callback*)
  - *URL* parameter specifies the URL you wish to load.
  - The optional *data* parameter specifies a set of query string key/value pairs to send along with the request.
  - The optional *callback* parameter is the name of a function to be executed after the load() method is completed.
- Reference

jQuery: AJAX – Get / Post

- Two commonly used methods for a request-response between a client and server:
  - **GET**: Requests data from a specified resource
  - **POST**: is used to send data to a server to create/update a resource.
  - HTTP Methods GET vs POST

jQuery: get() Method

- Syntax: $.get(*URL*, *callback*)
  - The *URL* parameter specifies the URL you wish to request.
  - The optional *callback* parameter is the name of a function to be executed if the request succeeds.
- Reference

jQuery: post() Method

- Syntax: $.post (*URL*, *data, callback*)
  - The *URL* parameter specifies the URL you wish to request.
  - The optional *data* parameter specifies some data to send along with the request.
  - The optional *callback* parameter is the name of a function to be executed if the request succeeds.
- Reference

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)

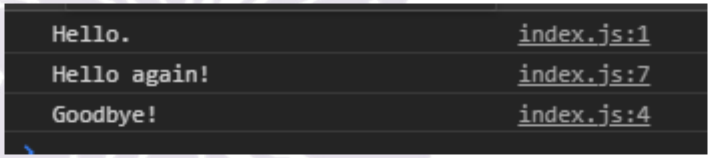
- We use asynchronous program to listen to events and then execute functions.
- After the event is triggered, some code will be executed, it’s called event handler.
- In the following case, “click” is the event fired, “console.log()” is the event handler.

```
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.
```



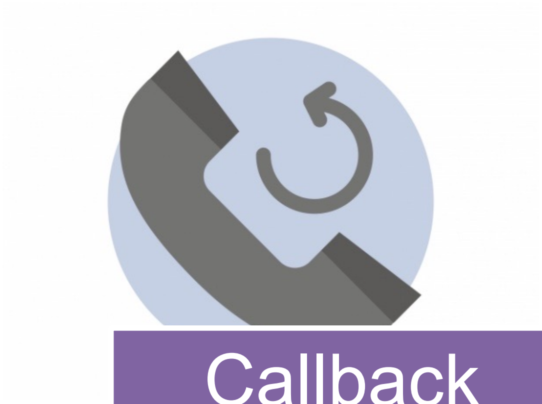
```
Hello.                          index.js:1
Hello again!                    index.js:7
Goodbye!                        index.js:4
```

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'!
```

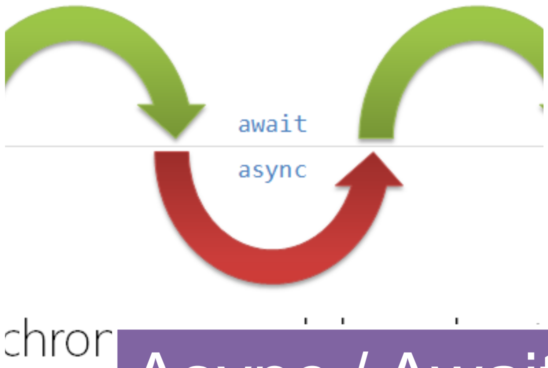
- In this example, if the image **img1** is not loaded before execute **addEventListener,** an error will occur.
- If you have a lot of images to be loaded in html, it will cause trouble.
- Thus, we need to handle the download process (or **img1**) **asynchronously**.

Three Approaches



Callback

Promise



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:

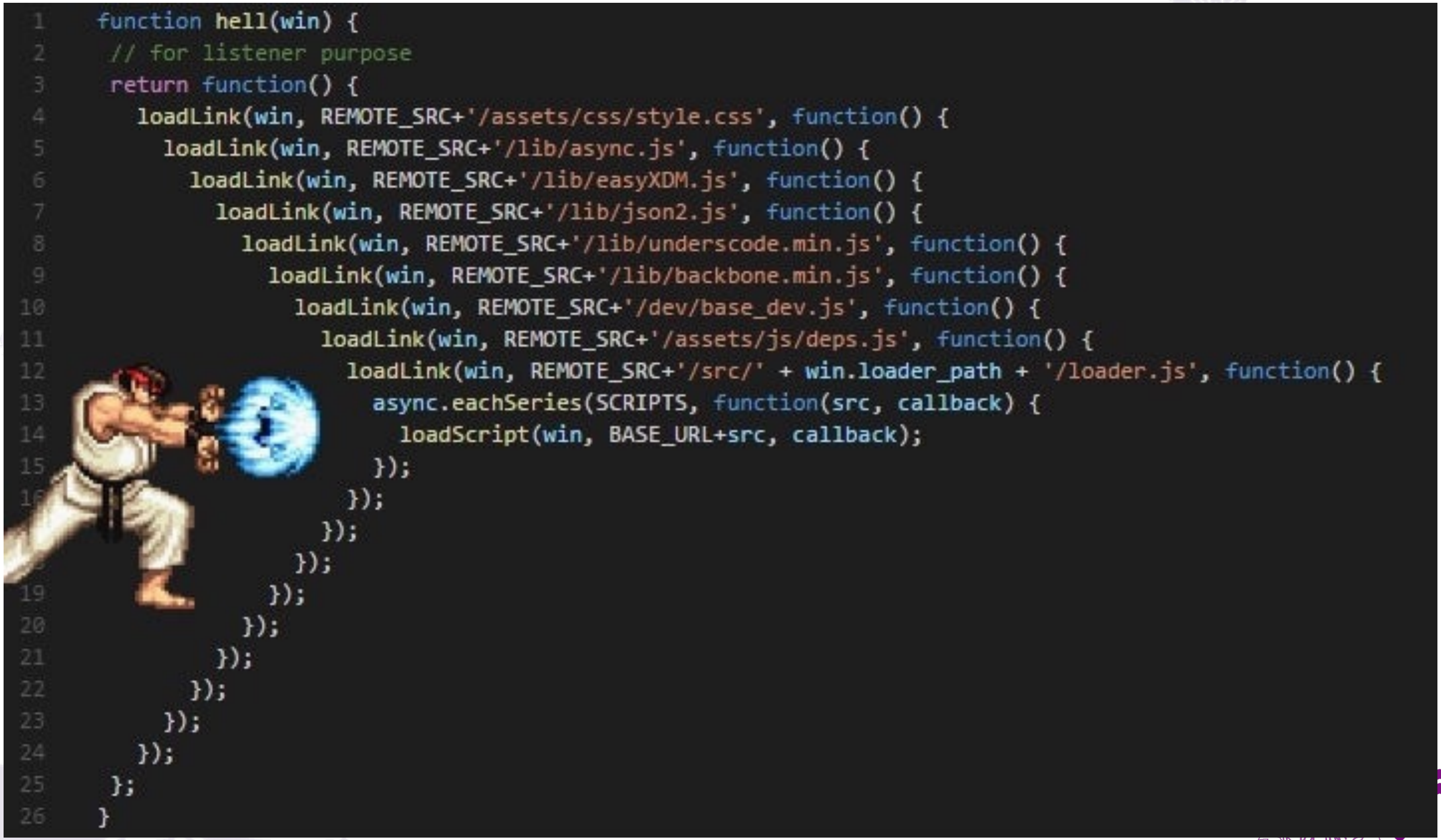
```
var text = readFile(fileName),
tokens = tokenize(text),
parseTree = parse(tokens),
optimizedTree = optimize(parseTree),
output = evaluate(optimizedTree);
console.log(output);
```

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)



```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

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) {
  evaluate(optimizedTree, evaluateFinish);
}
function evaluateFinish(output) {
  console.log(output);
}
readFile(fileName, readFinish);
```

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

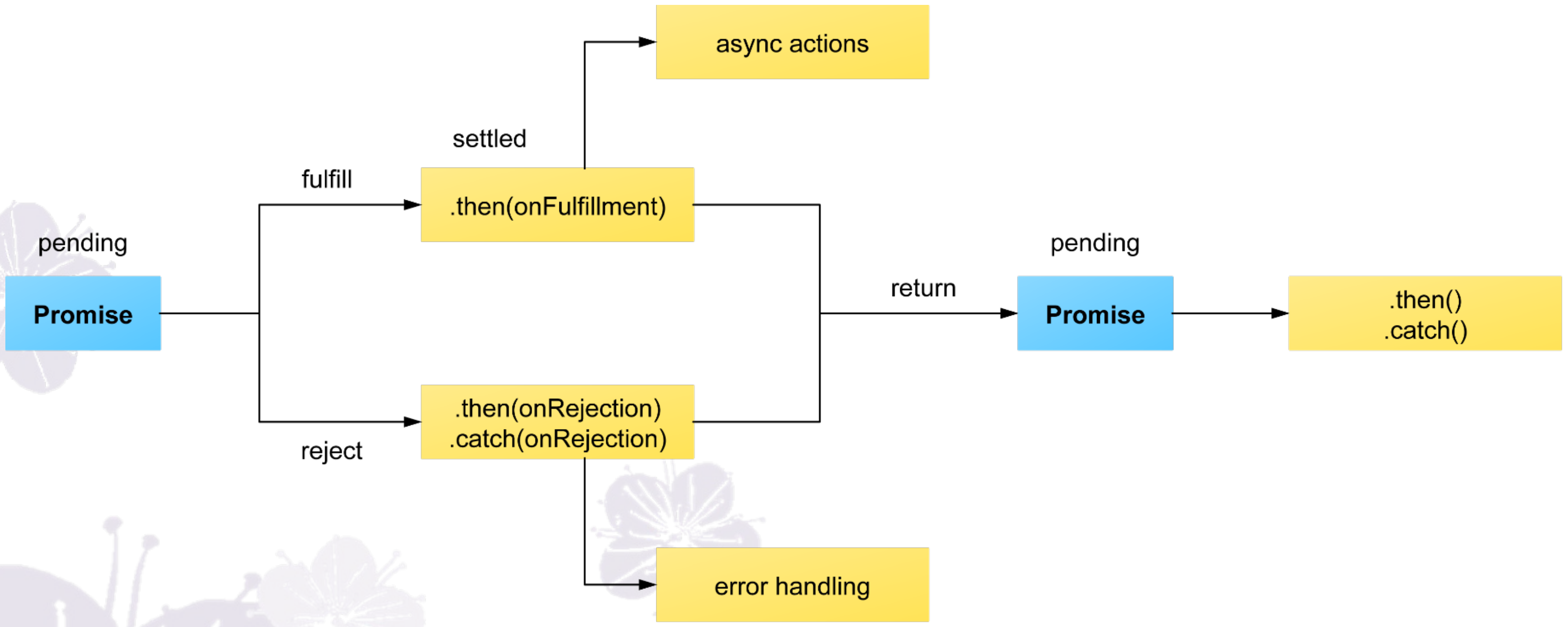
```
function successCallback(result) { console.log("Audio file ready at URL: " + result); }

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();
});
```

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

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

$\rightarrow$

```
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);
});
```

$\rightarrow$

```
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);
});
```

$\rightarrow$

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

.then() (Cont’d)

6. **Returns** by the handler. Also, the resolved value of the promise return **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 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);
  });
});
```

$\rightarrow$

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

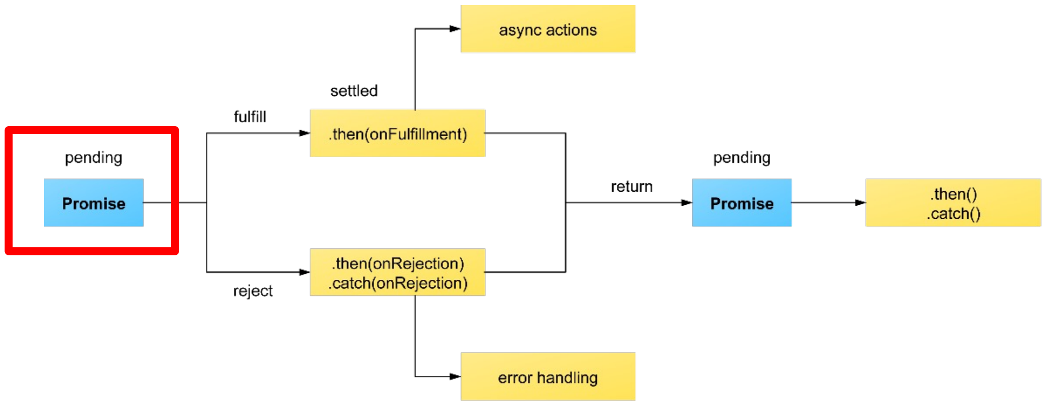
.catch()

- 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
});
```

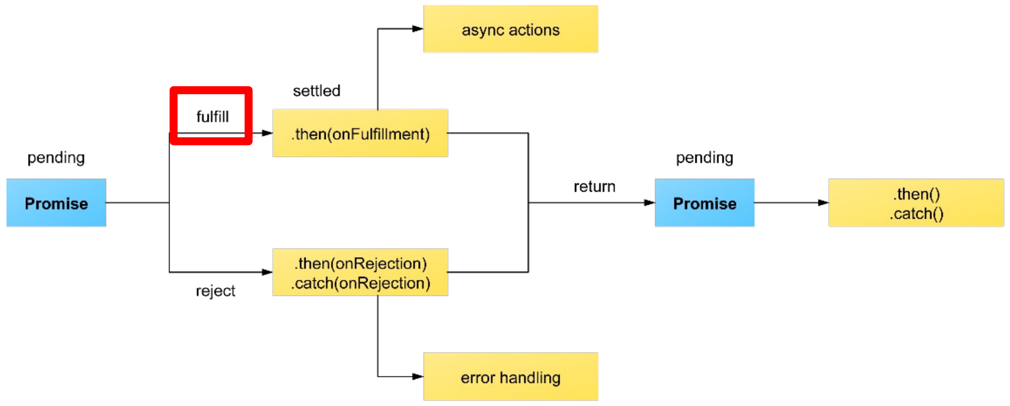
```
p.then(undefined, function(reason) {
  // handle the rejection
});
```

Chaining



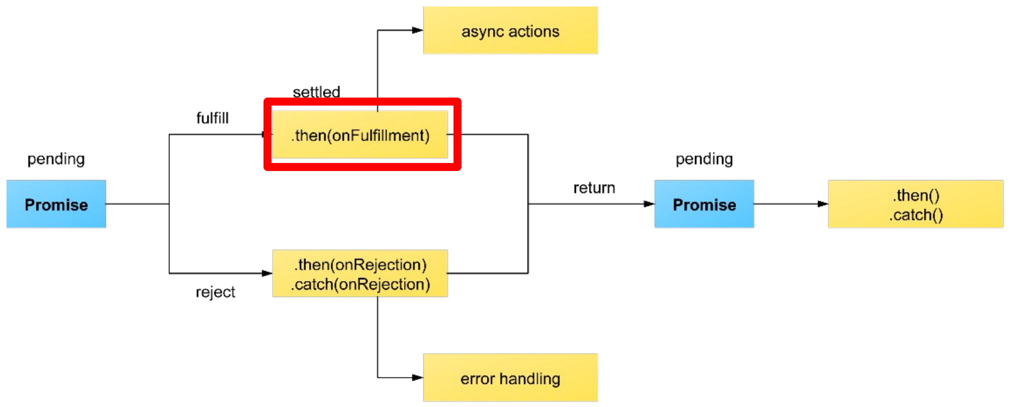
```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

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



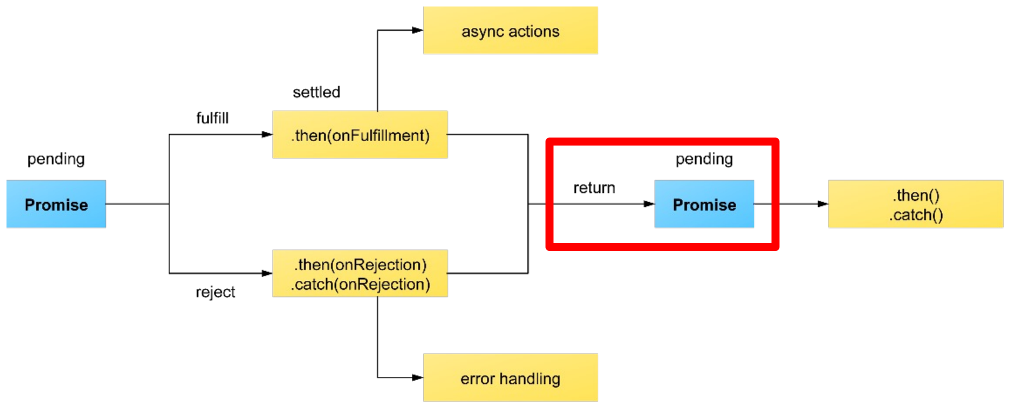
```
let p = new Promise(function(resolve, reject) {
 resolve(1);
});

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



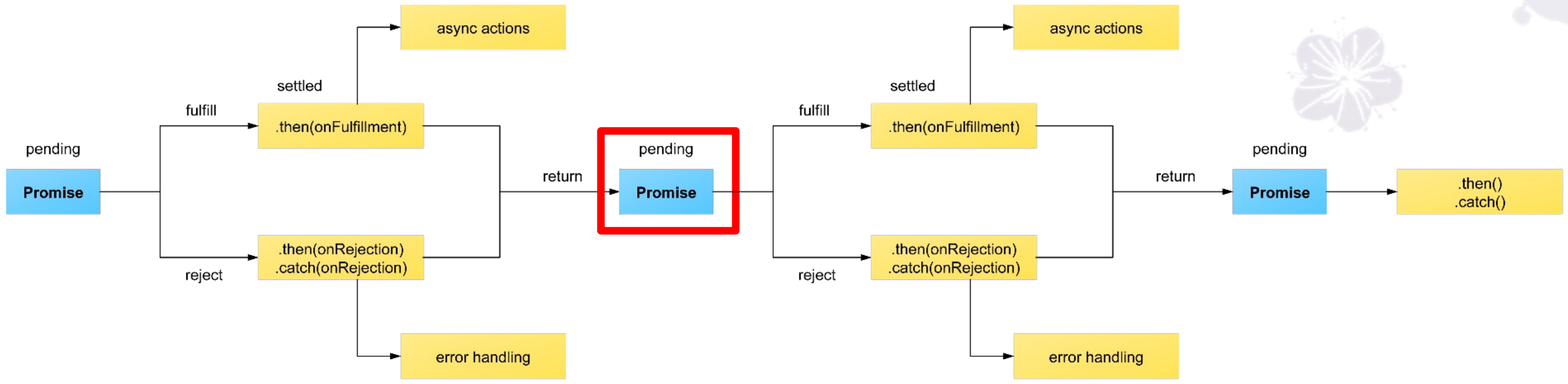
```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

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



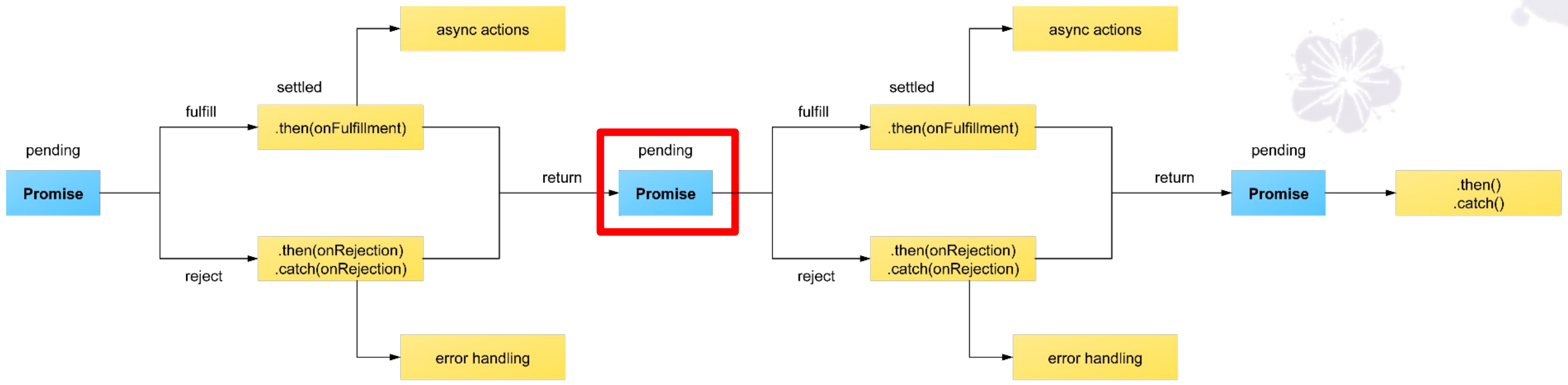
```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

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



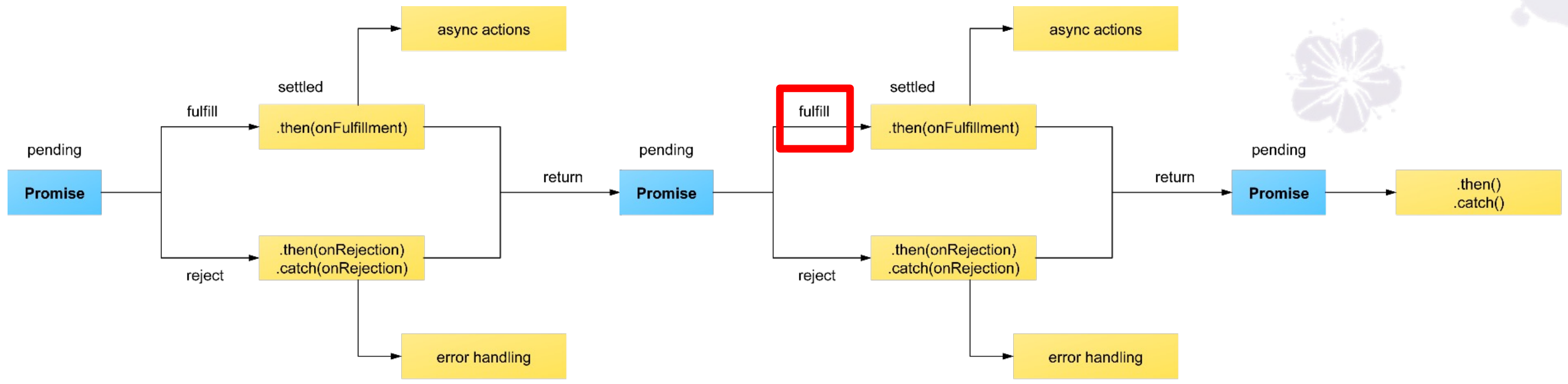
```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

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
});
```

Or you can write codes like these…

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

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

Chaining



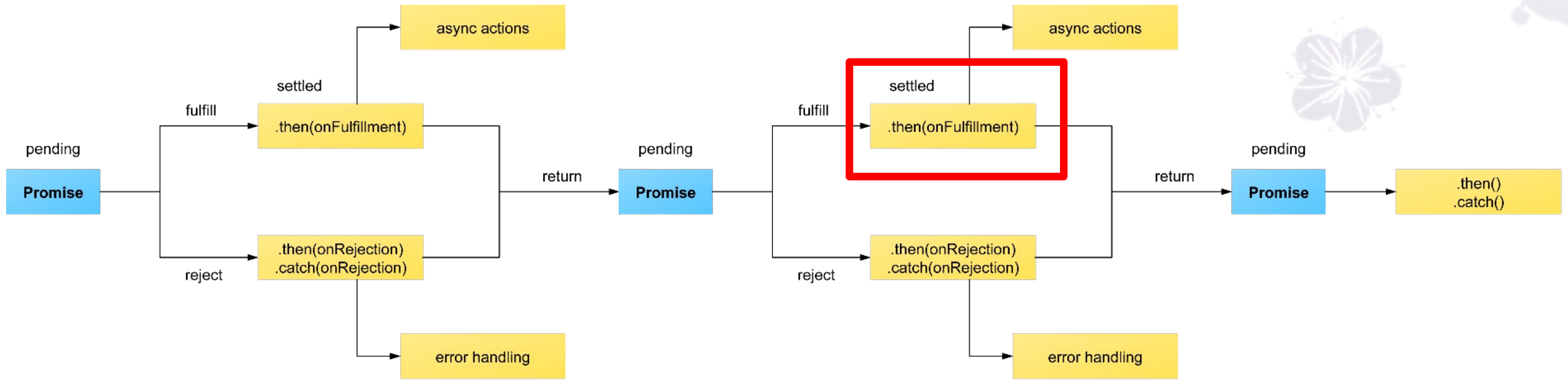
```
let p = new Promise(function(resolve, reject) {
  resolve(1);
});

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
});
```

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 <u>async promise-based</u> function will <u>pause the code until the promise fulfills/rejects.</u>
- **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

- The total duration is 2+2.5 = 4.5s

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!