

JavaScript

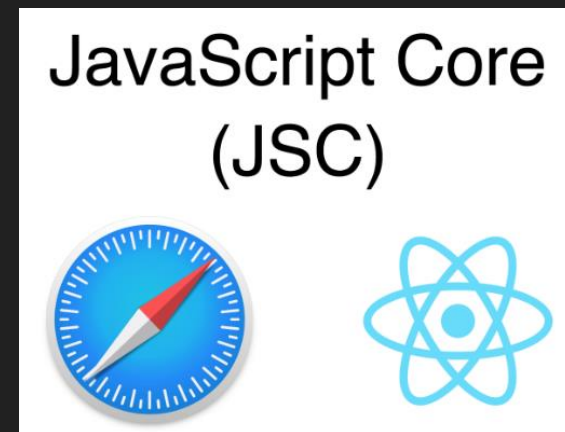
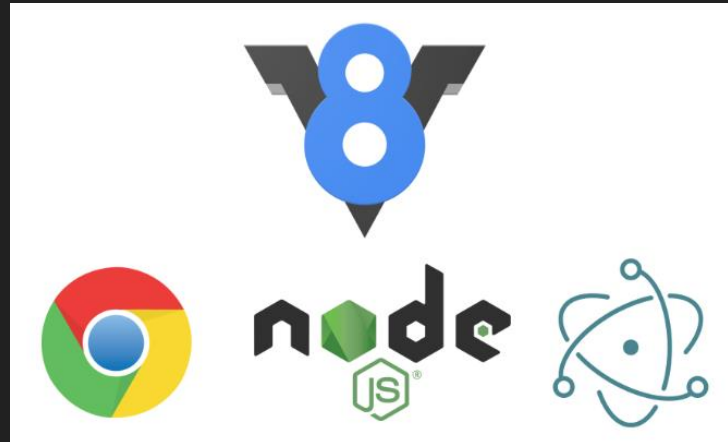
동기/비동기 – 김근형 강사

JavaScript Engine

- Javascript 엔진이란?
 - JS 코드를 실행하는 프로그램 또는 인터프리터를 말한다.
 - 자바 스크립트 코드를 생성하여 실행하게 되면 해당 엔진을 통해 자바스크립트가 실행된다.
 - 자바 스크립트 엔진은 상당히 여러가지가 있으며 그 중에 가장 중점적으로 다뤄야 할 엔진은 v8 엔진이다.
 - v8엔진은 구글 크롬과 node.js에 탑재된 엔진이며 현 html5에서 가장 주목받고 있는 엔진이기도 하다.

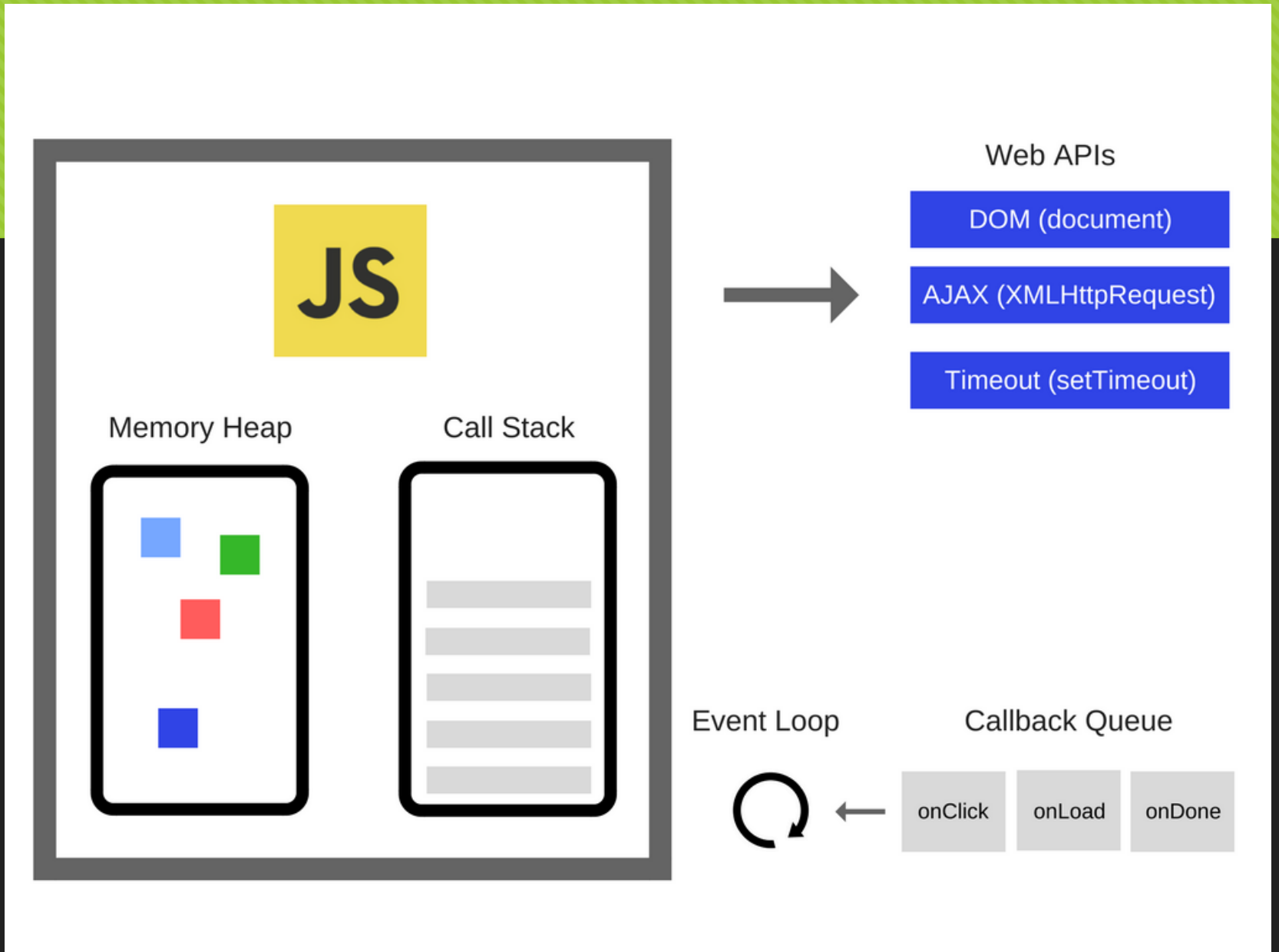
JavaScript Engine

○ Javascript 엔진종류



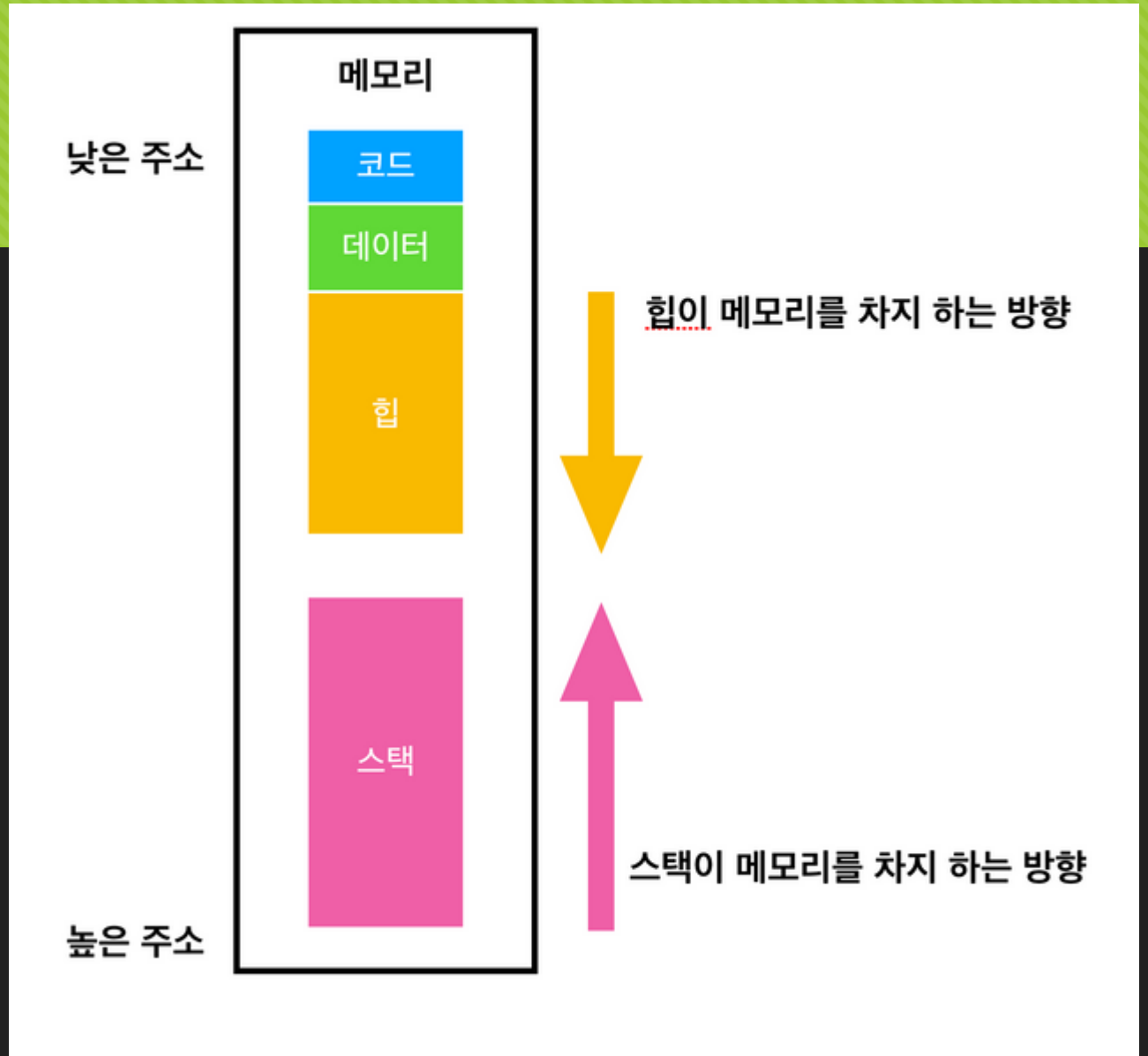
V8 Engine

○ V8 엔진의 구조



V8 Engine

- V8 엔진의 메모리 생성 방식



V8 Engine

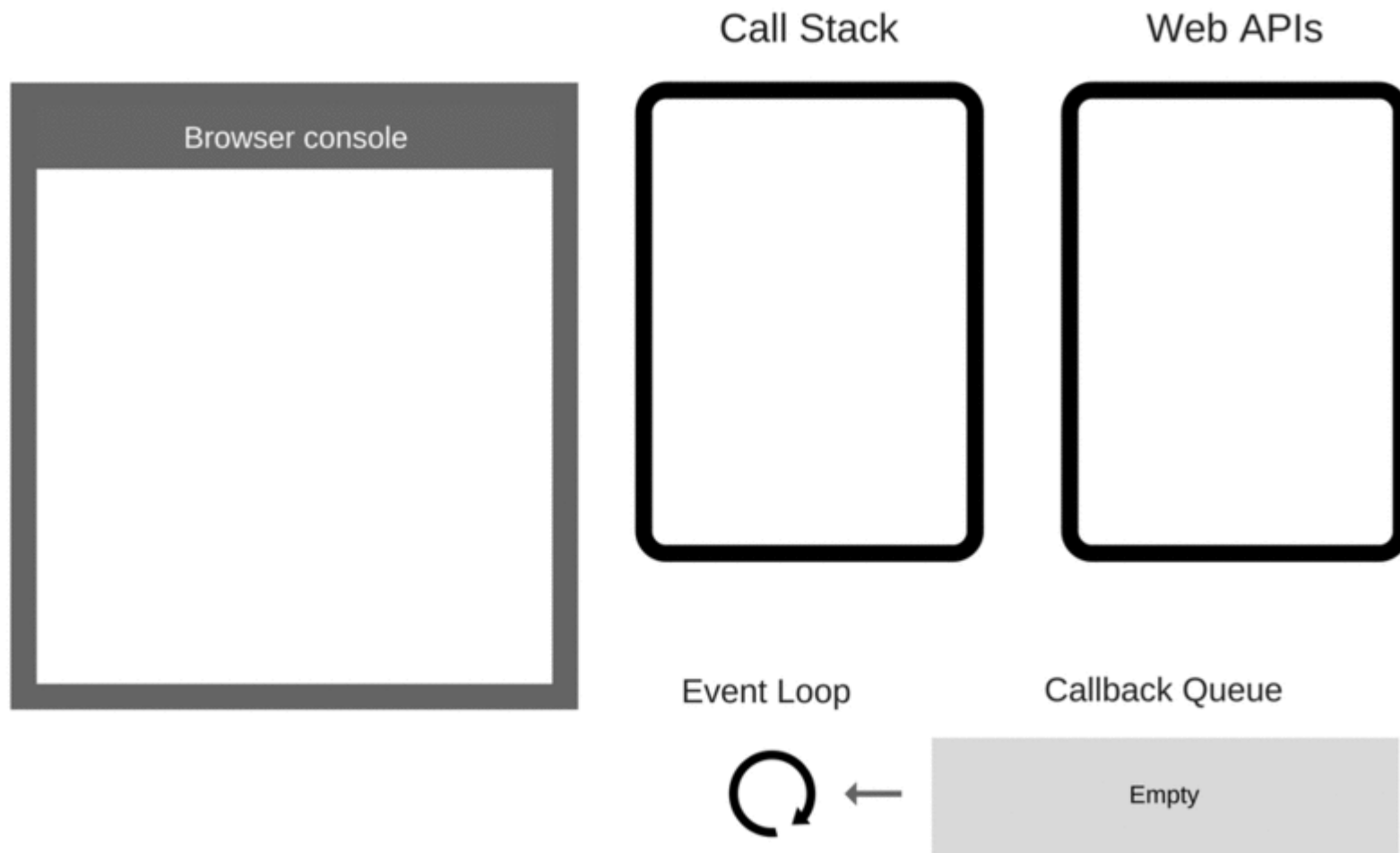
○ V8 메모리 영역 설명

- 코드영역 : 우리가 적은 코드가 적재되는 영역이다. 컴파일이 끝난 기계어로 들어간다고 한다.
- 데이터 영역 : 전역 변수 static 변수들이 적재되는 곳이다. 프로그램이 끝날 때 까지 존재한다.
- 스택 영역 : 지역변수, 매개 변수들이 저장되는데 함수 호출 시 사용되고 끝나면 반환된다. 함수가 호출될 때 그 메모리가 할당된다고 하여 콜 스택이라고 부른다. 선입후출의 자료구조이다. 컴파일 시에 크기가 결정된다.
- 힙 영역 : 메모리가 동적으로 할당되는 곳이다. 런타임시에 크기가 결정된다고 한다.

V8 Engine

○ V8 동작 원리

1 / 16



동기/비동기

○ 동기/비동기

- 동기란 어떤 코드가 위에서부터 아래까지 흐름대로 순서대로 실행됨을 의미한다.
- 즉, 실행되는 순서에 맞추어 코드가 실행되는 것을 동기라고 한다.
- 반면 비동기는 순서에 상관없이 실행되는 코드를 의미한다.
- 비동기 코드는 JavaScript 내에서 외부와의 통신, 딜레이 함수 등 여러가지 이유가 있다.
- JavaScript에서는 이런 비동기 통신을 동기화 시키기 위해 많은 기법들이 등장하였다.

동기/비동기

○ 동기/비동기 예제

```
// 동기
function syncFunction1() {
  console.log(1);
}

function syncFunction2() {
  console.log(2);
}

function syncFunction3() {
  console.log(3);
}
```

```
syncFunction1();
syncFunction2();
syncFunction3();
```

1

2

3

```
// 비동기
function asyncFunction1() {
  setTimeout(function() {
    console.log(1);
  }, 1000);
}

function syncFunction1() {
  console.log(2);
}

function syncFunction2() {
  console.log(3);
}
```

```
asyncFunction1();
syncFunction1();
syncFunction2();
```

2

3

undefined

1

Callback

- Callback 함수로 비동기 잡기
 - 이전에 배운 Callback 함수를 이용해 비동기를 동기화 시킬 수 있다.

```
function asyncFunction1(cb) {  
  setTimeout(function() {  
    console.log(1);  
    cb();  
  }, 1000);  
}
```

```
function syncFunction1(cb) {  
  console.log(2);  
  cb();  
}
```

```
function syncFunction2() {  
  console.log(3);  
}
```

```
asyncFunction1(function(asyncFunction1Result) {  
  syncFunction1(function(syncFunction1Result) {  
    syncFunction2();  
  })  
});
```

1

2

3

Callback

- Callback 함수의 단점
 - callback 함수를 너무 많이 쓰게 되면 가독성이 떨어지며 소스가 지저분해진다.
 - 이것을 [콜백 지옥]이라 부른다

```
function add(x, callback){
    let sum = x + x;
    console.log(sum);
    callback(sum);
}

add(3, function(result){
    add(result, function(result2){
        add(result2, function(result3){
            add(result3, function(result4){
                console.log("에너지 파")
            })
        })
    })
});
```

6

12

24

48

에너지 파

Callback

- Callback 함수의 단점
 - 이런 콜백 지옥을 막기 위해 각각의 함수를 명명하여 붙일 수도 있지만 재사용성이 떨어지고 소스가 길어져서 근본적인 해결책이 되지는 못했다.

```
function add1(x){  
  let sum = x + x;  
  console.log(sum);  
  add2(sum);  
}
```

```
function add2(x){  
  let sum = x + x;  
  console.log(sum);  
  add3(sum);  
}
```

```
function add3(x){  
  let sum = x + x;  
  console.log(sum);  
  add4(sum);  
}
```

```
function add4(x){  
  let sum = x + x;  
  console.log(sum);  
  add5(sum);  
}
```

```
function add5(x){  
  console.log("에너지 파")  
}
```

에너지 파 add1(1);

2

4

8

16

에너지 파

Callback

- 에러 처리의 한계
 - 콜백함수 내에서 예외를 발생시켰을 시 예외처리가 제대로 되지 않는다.

```
try {  
  setTimeout(() => { throw new Error('Error!'); }, 1000);  
} catch (e) {  
  console.log('에러를 캐치하지 못한다..');  
  console.log(e);  
}
```

7

▶ Uncaught Error: Error!
at <anonymous>:2:30

Promise

- Promise
 - Promise는 이러한 콜백 지옥의 단점을 보완하기 위해 ES6에서 제공한 기능이다.
 - 콜백 함수를 통해 나오는 코드의 복잡도를 좀 더 가독성 있게 만들 수 있다.
 - 또한 콜백 함수에서 일어나는 예외나 에러 처리가 가능하다는 점에서 상당히 좋은 이점을 가지고 있다.

Promise

- Promise 선언

- Promise를 사용하기 위해선 Promise 클래스 선언을 먼저 하여야 하며 선언은 아래와 같은 형태로 선언이 가능하다.

```
new Promise(function(resolve, reject) {  
  // Promise에서 처리할 코드  
});
```

- 여기서 Promise 클래스 선언 시 callback 함수를 인자로 가지게 되는데 해당 콜백 함수에서는 매개변수로 resolve 와 reject 란 매개변수를 가진다.

Promise

○ Promise 상태

- Pending(대기) : 비동기 처리 로직이 아직 완료되지 않은 상태, new Promise() 선언 시 Pending 상태가 된다.

```
new Promise(...);
```

- Fulfilled(이행) : 비동기 처리가 완료되어 프로미스가 결과 값을 반환해 준 상태, 콜백 함수의 인자 resolve를 실행하면 이행(Fulfilled) 상태가 된다. 이행 상태가 되면 then()을 이용하여 처리 결과 값을 받을 수 있다.

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


Promise

- Promise 상태

- Fulfilled(이행) : reject를 호출한 상태. reject를 호출하게 되면 실패 상태가 되며 실패한 이유(실패 처리의 결과 값)를 then() 혹은 catch()로 받을 수 있다.

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

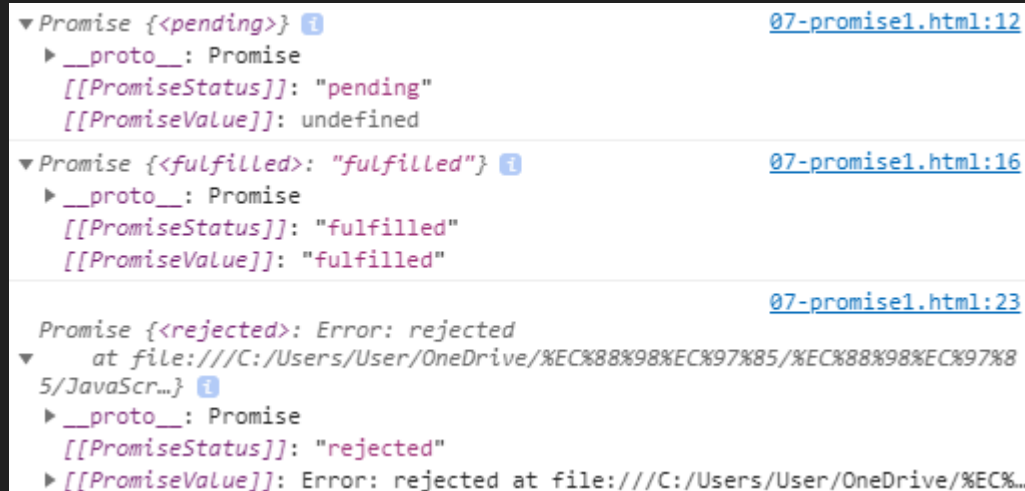
Promise

○ Promise 상태 예제 - 1

```
// pending 대기상태
var pending = new Promise((resolve) => {});
console.log(pending);

// fulfilled 성공상태
var fulfilled = new Promise((resolve) => resolve('fulfilled'));
console.log(fulfilled);

// rejected 실패상태
var rejected = new Promise((resolve, reject) => {
  throw new Error('rejected');
  reject();
});
console.log(rejected);
```

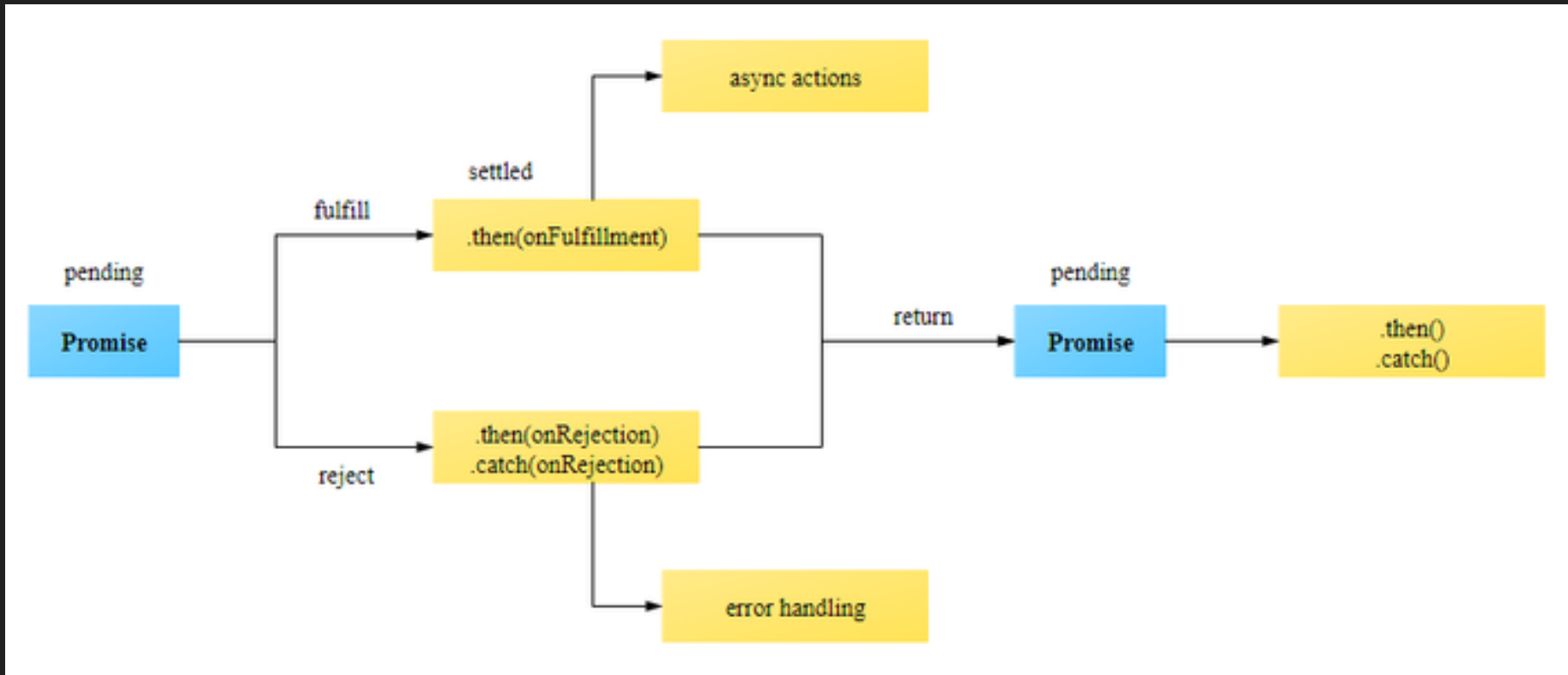


The screenshot shows a web browser's developer console with three Promise objects and their states:

- Promise {<pending>}** (07-promise1.html:12):
 - __proto__: Promise
 - [[PromiseStatus]]: "pending"
 - [[PromiseValue]]: undefined
- Promise {<fulfilled>: "fulfilled"}** (07-promise1.html:16):
 - __proto__: Promise
 - [[PromiseStatus]]: "fulfilled"
 - [[PromiseValue]]: "fulfilled"
- Promise {<rejected>: Error: rejected}** (07-promise1.html:23):
 - at file:///C:/Users/User/OneDrive/%EC%88%98%EC%97%85/%EC%88%98%EC%97%85/JavaScr... (07-promise1.html:23)
 - __proto__: Promise
 - [[PromiseStatus]]: "rejected"
 - [[PromiseValue]]: Error: rejected at file:///C:/Users/User/OneDrive/%EC%...

Promise

○ Promise 처리 흐름



Promise

- Promise.then()

- then() 메서드는 Promise를 리턴하고 두 개의 콜백 함수를 인수로 받는다. 하나는 Promise가 이행했을 때, 다른 하나는 거부했을 때를 위한 콜백 함수이다.

```
var p1 = new Promise(function(resolve, reject) {  
  resolve("성공!");  
  // 또는  
  // reject("오류!");  
});  
  
p1.then(function(value) {  
  console.log(value); // 성공!  
}, function(reason) {  
  console.log(reason); // 오류!  
});
```

Promise

- Promise.then()
 - then() 메서드는 체인 형태로 연속적으로 쓸 수가 있다. 이런 것을 체이닝이라고 한다.
 - then() 메서드에서 체인되는 then 메서드로 값을 보내기 위해서 resolve 로 보낼 수도 있고 return으로 보낼 수도 있다.
 - 단 return 시 비동기 함수의 처리 내용이 처리가 안될 수 있으니 주의할 것

```
Promise.resolve('foo')
// 1. "foo"를 받고 "bar"를 추가한 다음 그 값으로 이행하여 다음 then에 넘겨줌
.then(function(string) {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      string += 'bar';
      resolve(string);
    }, 1);
  });
})
// 2. "foobar"를 받고 그대로 다음 then에 넘겨준 뒤,
// 나중에 콜백 함수에서 가공하고 콘솔에 출력
.then(function(string) {
  setTimeout(function() {
    string += 'baz';
    console.log(string);
  }, 1)
  return string;
})
// 3. 이 부분의 코드는 이전의 then 블록 안의 (가짜) 비동기 코드에서
// 실제로 문자열을 가공하기 전에 실행됨
.then(function(string) {
  console.log("마지막 Then: 앗... 방금 then에서 프로미스 만들고 반환하는 걸 까먹어서 " +
    "출력 순서가 좀 이상할지도 몰라요");

  // 'baz' 부분은 setTimeout 함수로 비동기적으로 실행되기 때문에
  // 이곳의 string에는 아직 'baz' 부분이 없음
  console.log(string);
});
```

마지막 Then: 앗... 방금 then에서 프로미스 만들고 반환하는 걸 까먹어서 출력 순서가 좀 이상할지도 몰라요

foobar

foobarbaz

Promise

- Promise.then()

- then 핸들러에서 값을 그대로 반환한 경우에는 Promise.resolve(<핸들러에서 반환한 값>)을 반환하는 것과 같다.

```
var p2 = new Promise(function(resolve, reject) {
  resolve(1);
});

p2.then(function(value) {
  console.log(value); // 1
  return value + 1;
}).then(function(value) {
  console.log(value + ' - 동기적으로 짜도 돌아감');
});

p2.then(function(value) {
  console.log(value); // 1
});
```

1

1

2 - 동기적으로 짜도 돌아감

Promise

- Promise.then()

- 함수에서 오류가 발생하거나 거부한 프로미스를 반환한 경우 then에서는 거부한 프로미스를 반환한다.

```
Promise.resolve()  
.then(() => {  
    // .then()에서 거부한 프로미스를 반환함  
    throw new Error('으악!');  
})  
.then(() => {  
    console.log('실행되지 않는 코드');  
}, error => {  
    console.error('onRejected 함수가 실행됨: ' + error.message);  
});
```

onRejected 함수가 실행됨: 으악!

Promise

- Promise.then()

- 실제 개발 시에는 아래와 같이 거부한 프로미스를 then의 2단 핸들러 보다는 catch를 사용해 처리하는 경우가 많다.

```
Promise.resolve()  
  .then(() => {  
    // .then()에서 거부한 프로미스를 반환함  
    throw new Error('으악!');  
  })  
  .catch(error => {  
    console.error('onRejected 함수가 실행됨: ' + error.message);  
  })  
  .then(() => {  
    console.log("처음 then의 프로미스가 거부했지만 그래도 이 코드는 실행됨");  
  });
```


Promise

○ Promise.catch()

- promise 객체에서 reject를 통해 거부되거나 error가 발생되었을 경우 이벤트를 잡는 기능을 한다.
- then에서 이중 에러 처리를 하기보다 error를 통해 하게 되면 많은 then에서 일일이 처리하는 에러를 한꺼번에 처리가 가능하며 error 처리 후 then으로 다시 출력하는 것 또한 가능하다.

```
var p1 = new Promise(function(resolve, reject) {
  resolve('Success');
});

p1.then(function(value) {
  console.log(value); // "Success!"
  throw new Error('oh, no!');
}).catch(function(e) {
  console.error(e.message); // "oh, no!"
}).then(function() {
  console.log('after a catch the chain is restored');
}, function () {
  console.log('Not fired due to the catch');
});
```

```
var p1 = new Promise(function(resolve, reject) {
  resolve('Success');
});
// The following behaves the same as above
p1.then(function(value) {
  console.log(value); // "Success!"
  return Promise.reject('oh, no!');
}).catch(function(e) {
  console.error(e); // "oh, no!"
}).then(function() {
  console.log('after a catch the chain is restored');
}, function () {
  console.log('Not fired due to the catch');
});
```

Success
✖ ▶ oh, no!
after a catch the chain is restored

Promise

○ Promise.all()

- Promise.all 메소드는 프로미스가 담겨 있는 배열 등의 이터러블을 인자로 전달 받는다. 그리고 전달받은 모든 프로미스를 병렬로 처리하고 그 처리 결과를 resolve하는 새로운 프로미스를 반환한다.
- 여러 개의 비동기 작업들이 모두 완료되고 일을 진행시키고 싶을 때 사용한다.

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000)) // 3
]).then(console.log) // [ 1, 2, 3 ]
.catch(console.log);
```

Promise

- Promise.all()
 - Promise.all()은 배열 내 요소 중 어느 하나라도 거부하면 즉시 거부한다.

```
var p1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('하나'), 1000);
});
var p2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('둘'), 2000);
});
var p3 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('셋'), 3000);
});
var p4 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('넷'), 4000);
});
var p5 = new Promise((resolve, reject) => {
  reject(new Error('거부'));
});
```

```
// .catch 사용:
Promise.all([p1, p2, p3, p4, p5])
  .then(values => {
    console.log(values); // 출력안됨
  })
  .catch(error => {
    console.log(error.message) // 거부
  });
```

Promise

- Promise.all()
 - 심화예제

```
Promise.all(  
  [1, 2, 3]  
) .then(console.log)  
// [1, 2, 3]
```

```
Promise.all(  
  '123'  
) .then(console.log)  
// ["1", "2", "3"]
```

```
Promise.all([  
  Promise.resolve(1),  
  Promise.reject(2).catch(v => v),  
  Promise.resolve(3)  
) .then(console.log)  
// [1, 2, 3]
```

```
Promise.all([  
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(1) }, 4000)}),  
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(2) }, 2000)}),  
  new Promise((resolve, reject) => { setTimeout(() =>{ reject(3) }, 3000)}),  
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(4) }, 5000)}),  
  new Promise((resolve, reject) => { setTimeout(() =>{ reject(5) }, 1000)}),  
)  
) .then(list => { console.log(list); })  
  .catch(e => console.log(e));  
// 5
```

Promise

○ Promise.race()

- 전달된 Promise 들 중 가장 빨리 resolve 나 resolve 된 값을 반환한다.
- promise.race 는 iterable 값을 인자로 받기 때문에, Promise 가 아니더라도 래핑해서 Promise 로 처리한다.

```
Promise.race([
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(1) }, 2000)}),
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(2) }, 1000)}),
  new Promise((resolve, reject) => { setTimeout(() =>{ reject(3) }, 4000)}),
  new Promise((resolve, reject) => { setTimeout(() =>{ resolve(4) }, 3000)}),
  new Promise((resolve, reject) => { setTimeout(() =>{ reject(5) }, 5000)}),
])
.then(v => { console.log(v); }) // 2
.catch(e => console.log(e));
```

Async/Await

- `async/await`
 - `async`, `await` 는 ES8(ECMAScript2017)의 공식 스펙(링크)으로 비교적 최근에 정의된 문법
 - 기존의 `callback`과 `promise` 는 기존 코드를 읽기에 가독성이 그리 좋지 않았으며 직관적이지 못하다는 단점이 존재
 - ES8에서 `async`와 `await`을 도입했고, 덕분에 비동기 코드를 동기적으로 깔끔하게 처리할 수 있게 되었음
 - 하지만 콜백의 깊이가 깊지 않을 때는 작성하기 간편한 콜백함수를 호출하거나, `Promise`를 사용하는 것이 더 나을 수 있다.
 - `async/await`은 `Promise`를 사용하기 때문에 `Promise`를 알아야 하고, `async/await`이 할 수 없는 동작을 `Promise`로 해결할 수 있는 경우도 있다.

Async/Await

○ async/await 사용방법

```
async function name([param[, param[, ... param]]]) {  
    ...  
    [rv] = await expression;  
    ...  
}
```

- async는 비동기 기능이 존재하는 function 앞에 선언한다. 해당 function 내에서 동작하는 모든 비동기 function 을 동기식으로 동작하겠다는 범위의 지정이기도 하다.
- await 로 선언할 수 있는 대상은 promise 객체 이거나 값이 될 수 있으며 그 이외의 것들은 들어올 수 없다.
- await는 async 함수 내에서만 유효하다.

Async/Await

○ async/await 사용 기본 - 1

```
// async 함수는 내부적으로 Promise.resolve를  
// 리턴하는 것 처럼 동작하기 때문에 위의 함수는  
// 아래의 함수와 동일한 동작을 한다.
```

```
async function f() {  
    return 1;  
}
```

```
// async function f() {  
//   return Promise.resolve(1);  
// }
```

```
f().then(alert); // 1
```

```
// 프로미스 객체를 선언하여 들어가는 형태  
function resolveAfter2Seconds(x) {  
    return new Promise(resolve => {  
        setTimeout(() => {  
            resolve(x);  
        }, 2000);  
    });  
}
```

```
async function f1() {  
    var x = await resolveAfter2Seconds(10);  
    console.log(x); // 10  
}
```

```
f1();
```


Async/Await

○ async/await 사용 기본 - 2

// await를 연속적으로 호출하였을 경우
// Promise의 then과 같은 효과를 낼 수 있다.

```
async function test(){  
  await foo(1, 2000)  
  await foo(2, 500)  
  await foo(3, 1000)  
}
```

```
function foo(num, sec){  
  return new Promise(function(resolve, reject){  
    setTimeout(function(){  
      console.log(num);  
      resolve("async는 Promise방식을 사용합니다.");  
    }, sec);  
  });  
}  
test(); // 1,2,3
```

1

2

3

// await 앞에 있는 함수가 promise를 객체가
// 아닐 경우 await의 효과를 전혀 볼 수가 없다.

```
async function test(){  
  await foo(1, 2000)  
  await foo(2, 500)  
  await foo(3, 1000)  
}
```

```
function foo(num, sec){  
  setTimeout(function(){  
    console.log(num);  
  }, sec);  
}  
test();
```

2

3

1

Async/Await

○ async/await 사용 기본 - 3

```
// 만약 값이 Promise가 아닌 일반 리터럴 값일 경우
// 해당 값은 resolve된 Promise로 변환되며 이를 기다린다.
async function f2() {
  var x = await 20;
  var y = await (x + 10);
  console.log(y); // 30
}
f2();
```

```
// 만약 Promise가 reject되면, reject된 값이 throw된다.
async function f3() {
  try {
    var z = await Promise.reject(30);
  } catch(e) {
    console.log(e); // 30
  }
}
f3();
```

Async/Await

○ async/await 사용 예제



lurks in the shadows

```
function who() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('🐱');
    }, 200);
  });
}

function what() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('lurks');
    }, 300);
  });
}

function where() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('in the shadows');
    }, 500);
  });
}

async function msg() {
  const a = await who();
  const b = await what();
  const c = await where();

  console.log(`${ a } ${ b } ${ c }`);
}

msg();
```

Script tag 지원

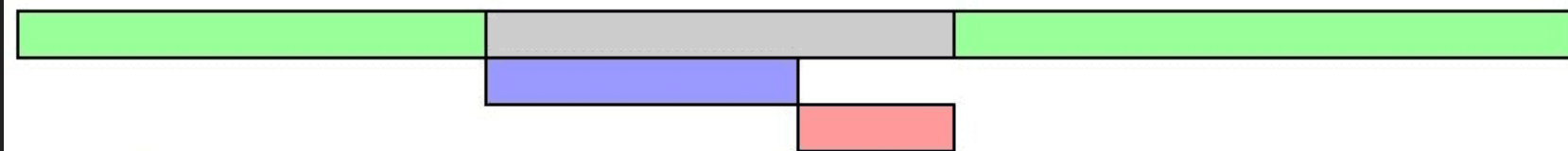
- Script tag 지원

- 스크립트 태그는 `<script>` 라는 태그를 통해 html에 외부의 javascript 파일을 가져올 수 있다.
- ECMAScript6 이전에는 html과 자바 스크립트 파일과의 동기화를 진행하기 위해 스크립트 태그를 html 파일의 맨 끝에 넣거나 자바 스크립트의 이벤트인 `window.onload()` 혹은 `window.DOMContentLoaded()` 이벤트를 써서 html 파일의 로드가 끝나면 불러올 수 있도록 세팅하였다.
- 하지만 이런 이벤트를 쓰는 것은 상당히 번거로웠으며 실시간으로 DOM으로 html 객체와 연결을 해야하는 경우 코드의 복잡도도 상당히 올라갔다.
- 이후 W3C 에서는 `<script>` 태그에 `async` 와 `defer` 라는 속성을 지원하게 된다.

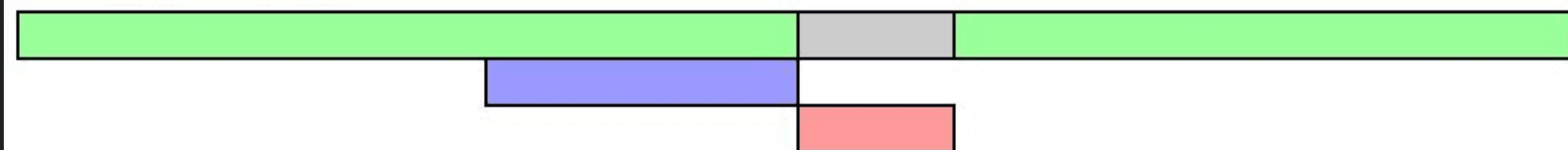
Script tag 지원

○ Script tag 지원

`<script>`



`<script async>`



`<script defer>`



Legend

HTML parsing HTML parsing paused Script download Script execution

Script tag 지원

- Script tag 지원

- `<script>` : html 태그를 읽는 중 `<script>` 태그를 만나게 되면 기존의 html 태그 읽는 것을 중지하고 스크립트를 다운로드 후 실행 후에 html 태그를 다시 읽어온다.
- `<script async>` : html 태그를 읽는 중 `<script async>` 태그를 만나게 되면 기존의 html 태그 읽는 것을 javascript 소스를 다운로드 하는 것과 동시에 진행된다. 단 javascript 실행시에는 html 코드 읽기가 중지된다.
- `<script defer>` : html 태그를 읽는 중 `<script defer>` 태그를 만나게 되면 기존의 html 태그 읽는 것을 javascript 소스를 다운로드 하는 것과 동시에 진행된다. 단 javascript 실행은 html 코드가 완전히 다 읽혀진 후에 실행된다.

Script tag 지원

○ Script tag 지원 예제

```
document.getElementById("btn").addEventListener("click", ()=>{
    document.getElementById("str").innerHTML = "hello"
}, false);
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <!-- <script type="text/javascript" src="./27-asyncdefer.js"></script> -->
  <!-- <script async type="text/javascript" src="./27-asyncdefer.js"></script> -->
  <script defer type="text/javascript" src="./27-asyncdefer.js"></script>
</head>
<body>
  
  <p id="str"></p><br>
  <button id="btn" type="button">클릭</button>
</body>
</html>
```



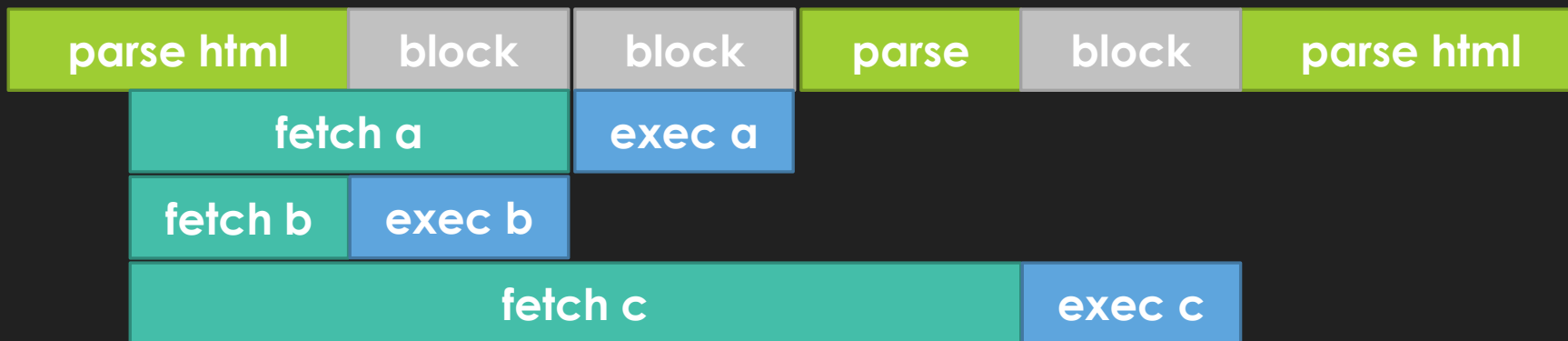
hello

클릭

Script tag 지원

- Script tag 지원

- `<script async>`의 경우 복수의 자바스크립트 파일을 실행시킬 시 각각의 자바스크립트 파일들은 독립적으로 다운로드 되며 실행된다.
- `<script async>`는 자바스크립트 파일의 실행 순서에 상관이 없고 DOM 오브젝트를 이용한 객체 이벤트 처리 같은 내용을 하지 않을 경우 적합하다.
- `<script async>` 를 통해 파일을 각기 실행시킬 경우 다음과 같은 실행 순서를 가진다.



Script tag 지원

- Script tag 지원
 - <script defer>의 경우 자바스크립트 파일을 읽어오는건 제각각이지만 실행은 결국 html 파일이 전부 읽혀진 후에 실행된다
 - 그렇기에 <script defer>는 html 소스의 head 태그 부분에 선언해도 크게 상관이 없다.