АСИНХРОННЫЙ JS

Tinkoff.ru

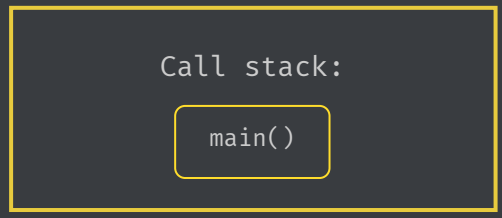
- Однопоточность
- Стек вызовов
- Очередь задач
- Асинхронные API

JS - ОДНОПОТОЧНЫЙ ЯЗЫК

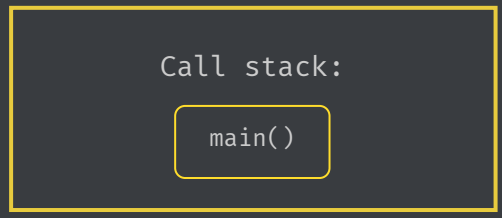
СТЕК ВЫЗОВОВ

MDN

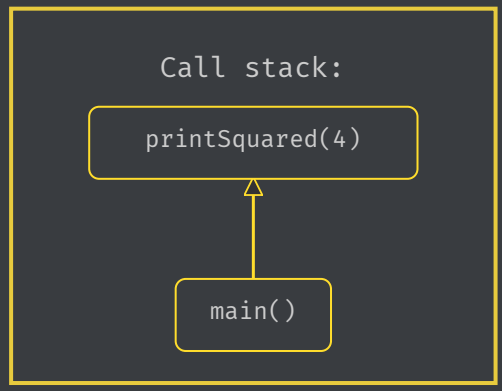
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



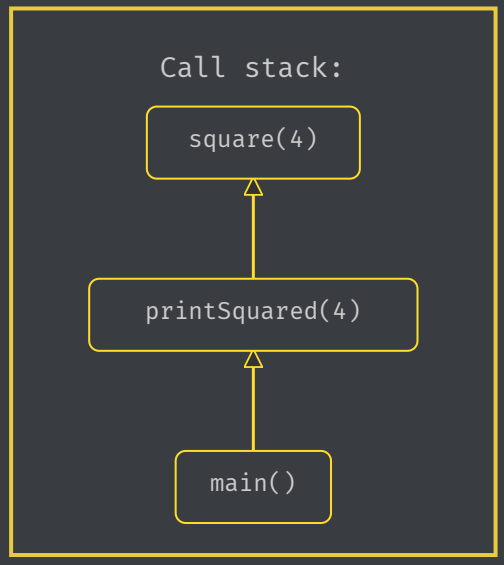
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



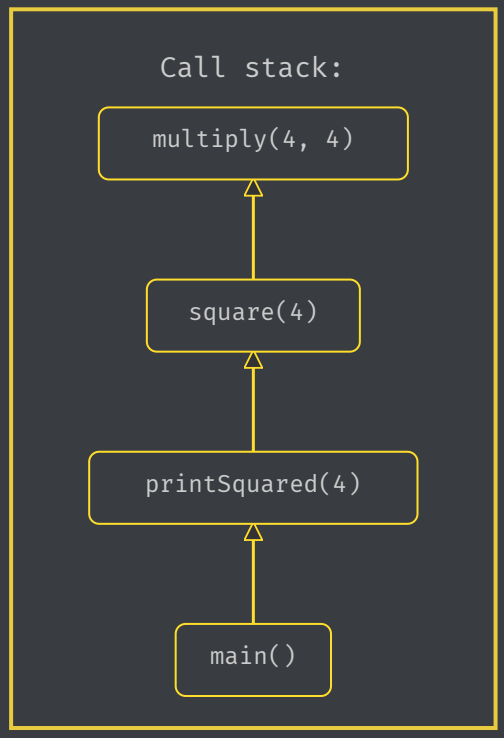
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



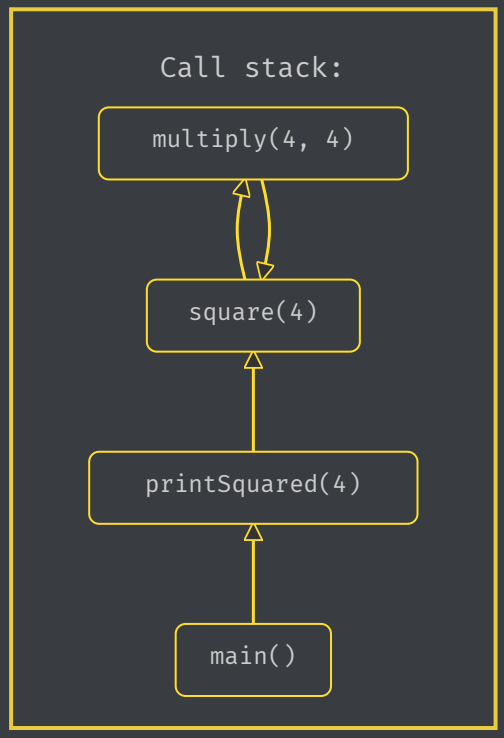
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



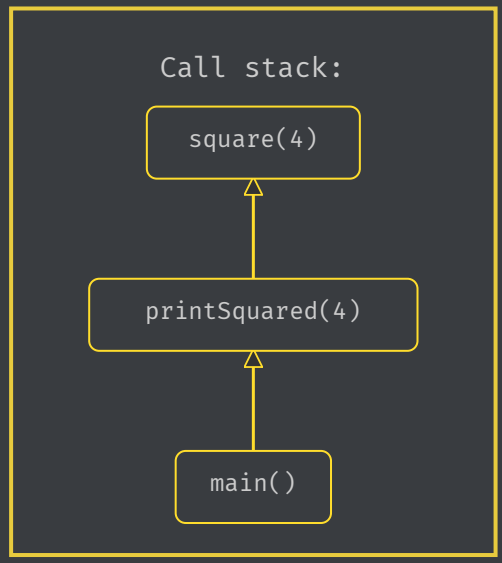
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



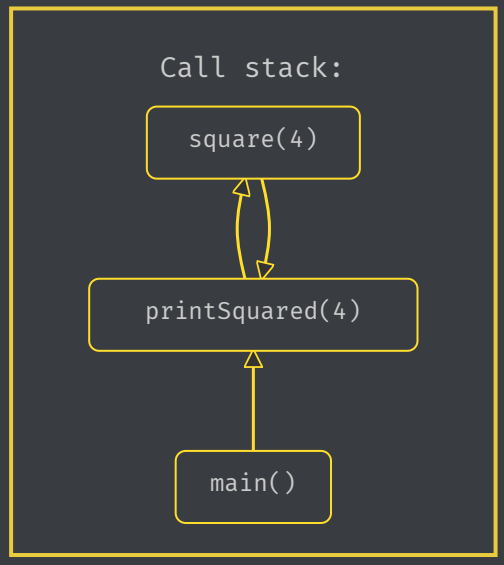
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



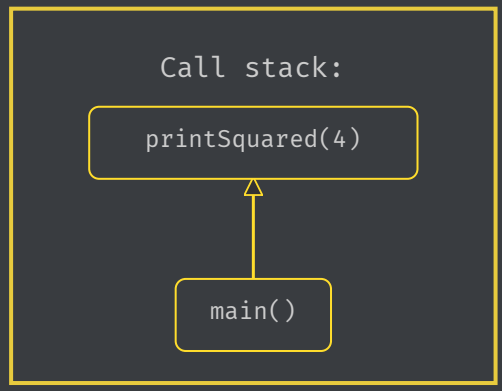
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



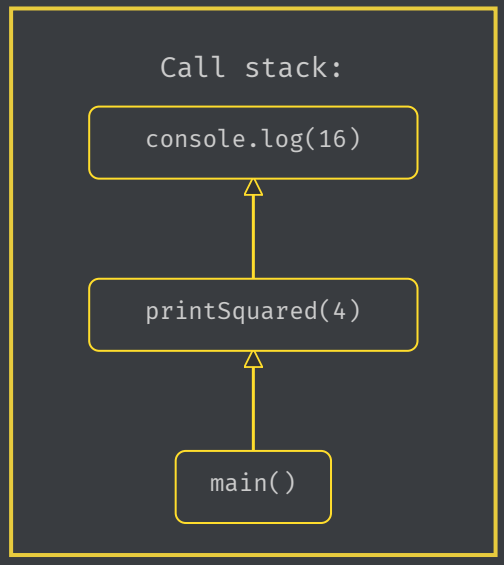
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



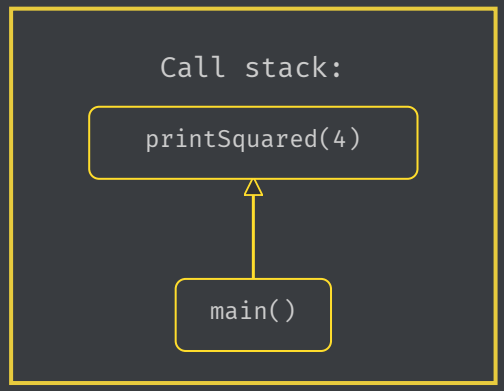
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



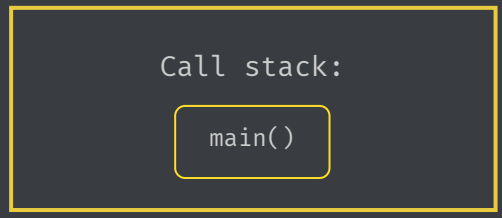
```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```



```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```

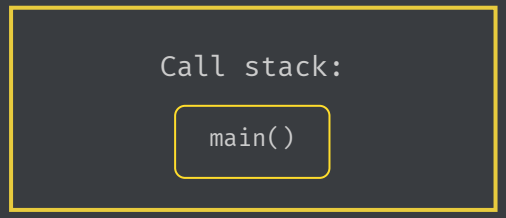


```
function multiply(a, b) {
  return a * b;
}

function square(x) {
  return multiply(x, x);
}

function printSquared(x) {
  const squared = square(x);
  console.log(squared);
}

printSquared(4);
```

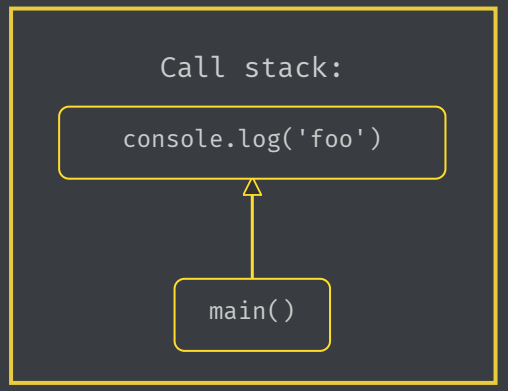
Call stack:
‹empty›

Call stack — это "last in, first out"

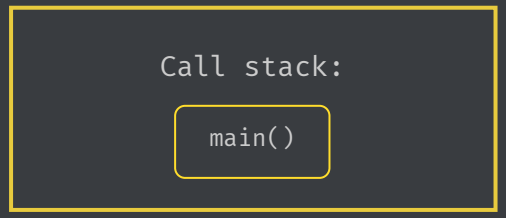
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



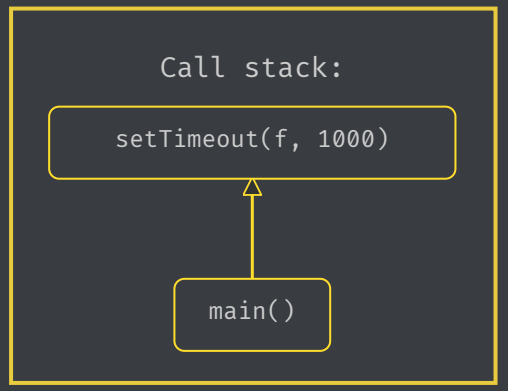
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



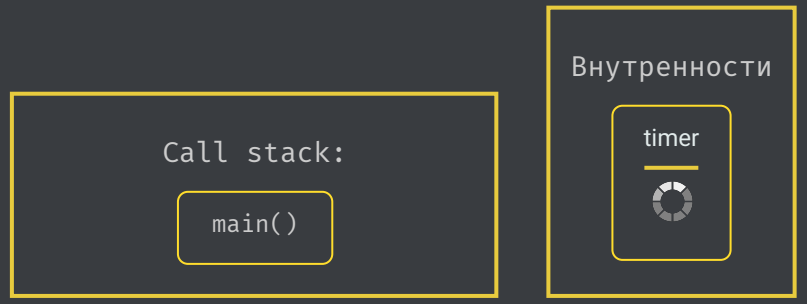
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



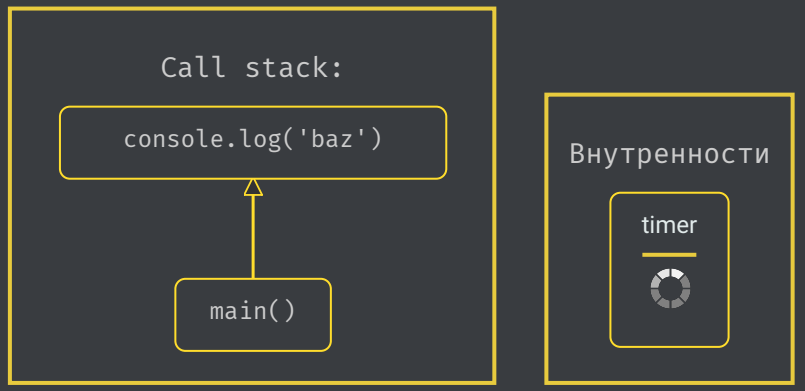
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



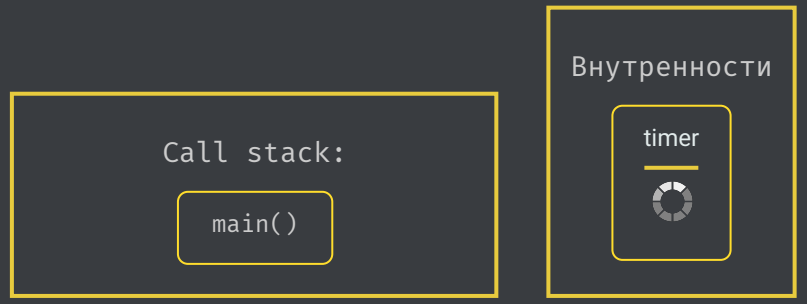
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



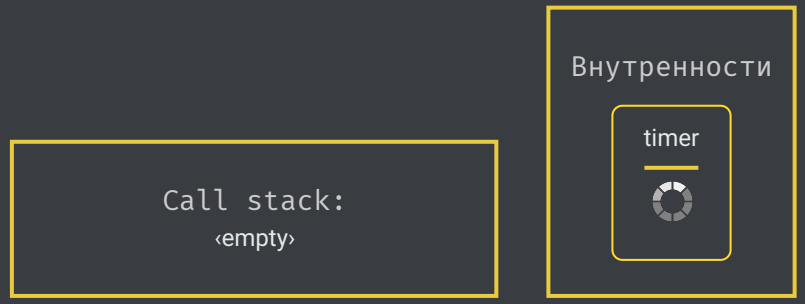
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



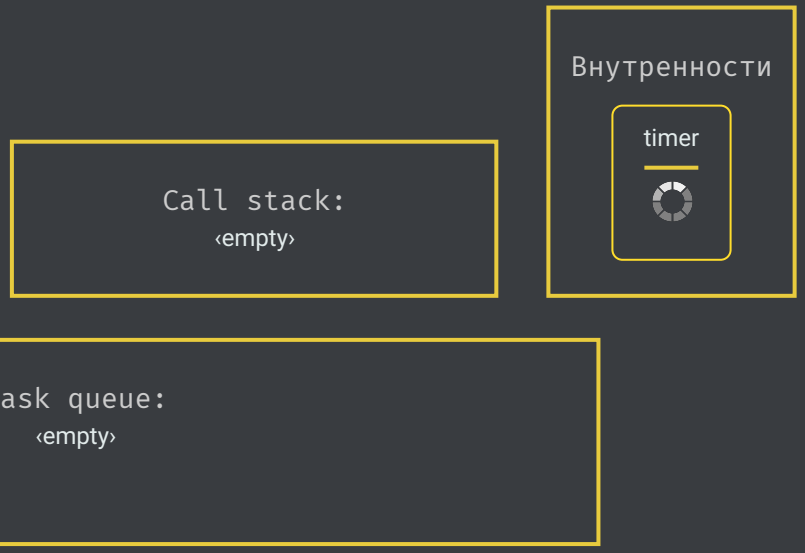
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



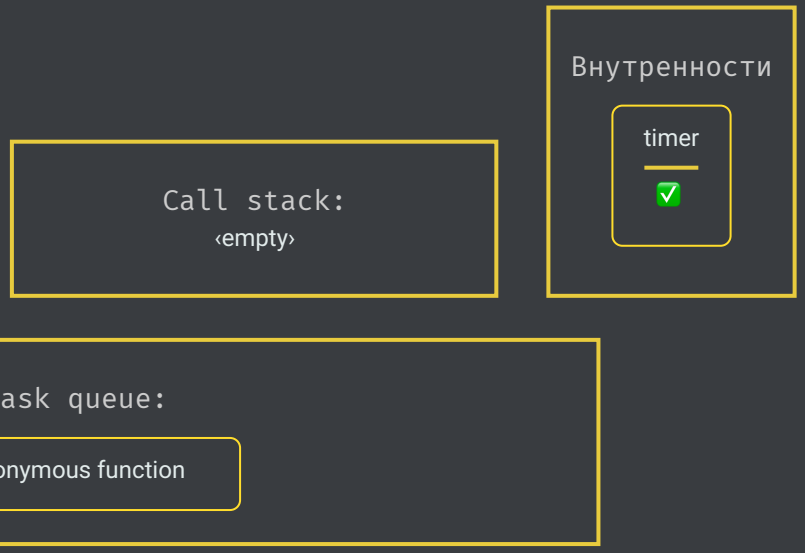
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



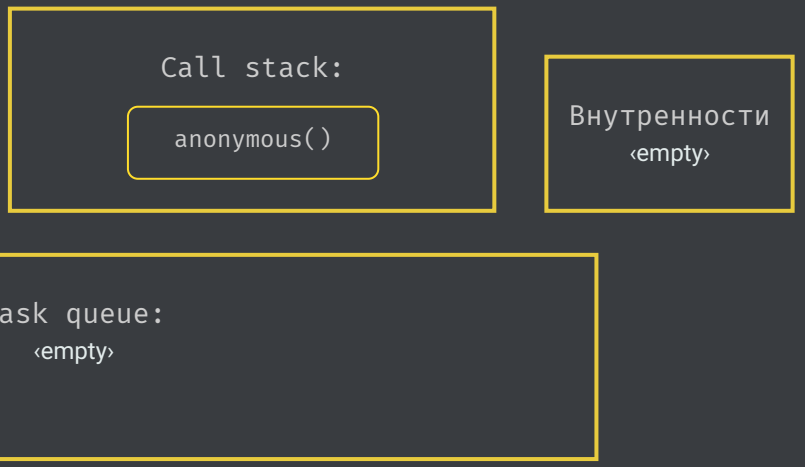
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



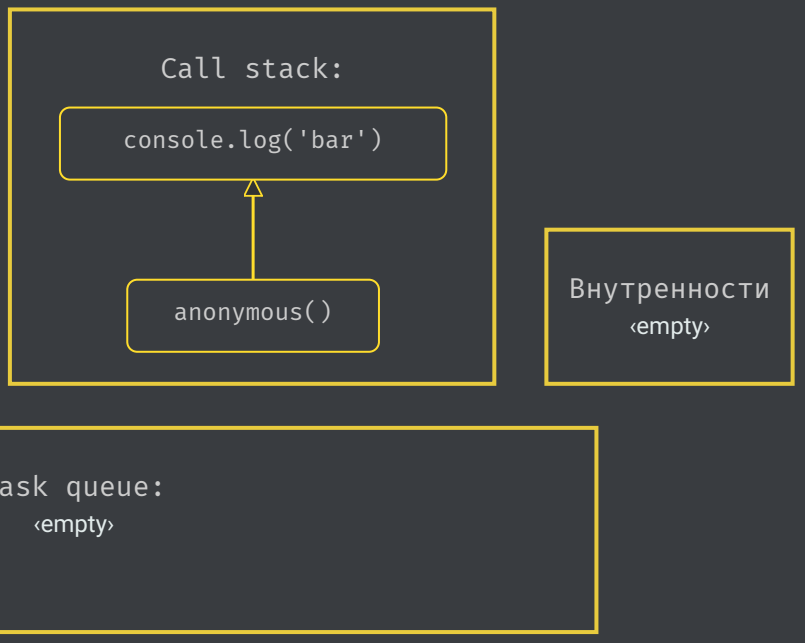
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



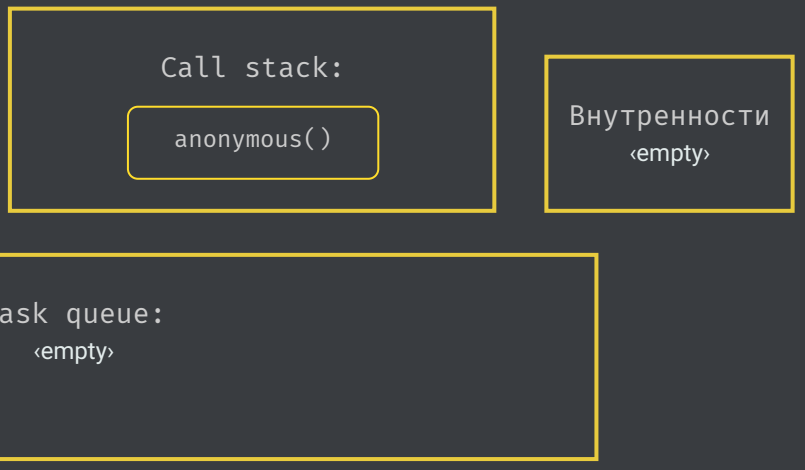
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



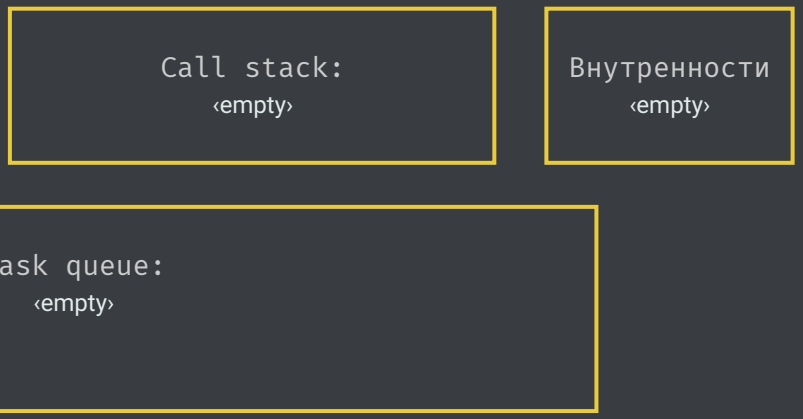
```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



```
console.log('foo');

setTimeout(
  function () {
    console.log('bar');
  },
  1000
);

console.log('baz');
```



Task queue — это "last in, last out"

```
while (taskQueue.waitForTask()) {
  taskQueue.executeNextTask();
}
```

```
setInterval(() => {
  console.log('foo');
}, 1000);
```

Другие API, откладывающие вызов коллбека в новый task.

```
setImmediate(() => {
  console.log('foo');
});
```

setImmediate есть только в Node.js и IE.

```
document.body.addEventListener('click', () => {
  console.log('clicked');
})
```

```
const xhr = new XMLHttpRequest();
```

XHR нужен для ЗАПРОСОВ (внезапно)

XHR Example

ИТОГО

- Синхронные запросы: НЕЛЬЗЯ! ПЛОХО! АЛАРМ! ТАБУ! ПРОХИБИТЕД! ЗАПРЕЩЕНО! ФЕРБОТЕН!
- Асинхронные запросы: ок

PROMISE

learn.javascript.ru

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

CALLBACK HELL

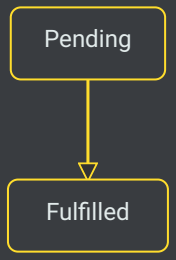
```
makeRequest(url1, someData, function (result1) {
  makeRequest(url2, result2, function (result2) {
    makeRequest(url3, result3, function (result3) {
      var result4 = null;
      var result5 = null;

      function next () {
        makeRequest(url6, result4, result5, function (result6) {
          makeRequest(url7, result6, function (result7) {
            doSmthWith(result7);
          })
        })
      }

      makeRequest(url4, result3, function (res) {
        result4 = res;
        if (result5) next();
      })

      makeRequest(url5, result3, function () {
        result5 = res;
        if (result4) next();
      })
    })
  })
})
```

ТО ЖЕ, НО НА ПРОМИСАХ

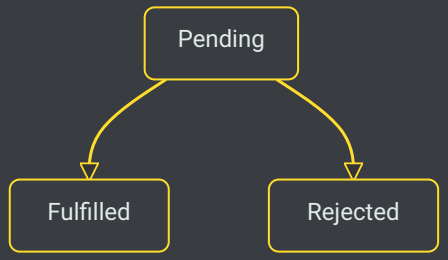
```
makeRequest(url1, someData)
  .then(result => makeRequest(url2, result))
  .then(result => makeRequest(url3, result))
  .then(result => Promise.all([
    makeRequest(url4, result),
    makeRequest(url5, result)
  ])
  .then(([r1, r2]) => makeRequest(url6, r1, r2))
  .then(result => makeRequest(url7, result)
  .then(result => doSmthWith(result))
```

Pending

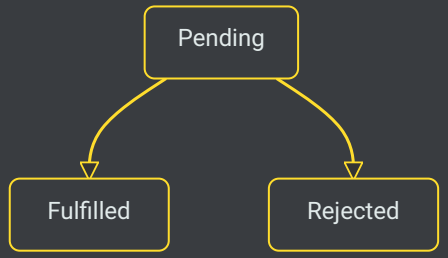
```
new Promise((resolve, reject) => {

});
```




```
new Promise((resolve, reject) => {
  resolve('foo'); // При успехе
});
```




```
new Promise((resolve, reject) => {
  reject(new Error('bar')); // При ошибке
});
```



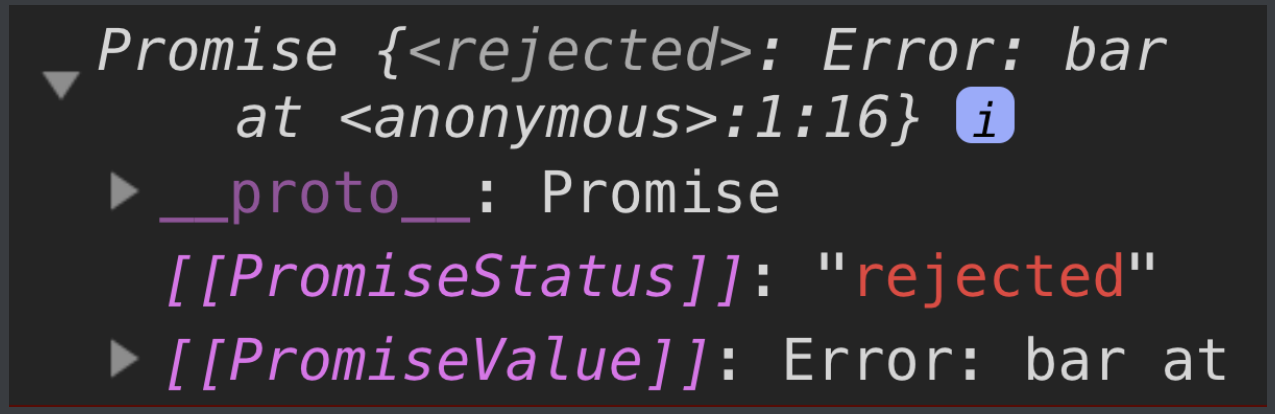
Promise может иметь одно из трех состояний.

```
Promise.resolve('foo');
```

```
new Promise(resolve => {
  resolve('foo');
})
```

```
Promise.reject(
  new Error('bar')
);
```

```
new Promise((_, reject) => {
  reject(
    new Error('bar')
  );
})
```



Promise.prototype.then()

```
Promise.resolve(10)
  .then(x => x * 2);

// Promise { <fullfiled>: 20 }
```

```
Promise.resolve(10)
  .then(x => x * 2)
  .then(x => x / 5)
  .then(x => x.toString())

// Promise { <fullfiled>: '4' }
```

```
Promise.reject(10)
  .then(
    x => x * 2,
    error => console.log(error)
  )

// Promise { <fullfiled>: undefined }
```

Второй аргумент ловит ошибку.

```
Promise.resolve('foo')
  .then(
    x => {
      throw 'error!';
    }
  )

// Promise { <rejected>: 'error!' }
```

```
Promise.reject('bar')
  .then(
    null,
    err => {
      throw 'error!';
    }
  )

// Promise { <rejected>: 'error!' }
```

NB: если в любом из коллбеков упала ошибка, вернется Promise в состоянии rejected.

```
Promise.resolve('foo')
  .then(() => Promise.resolve('bar'))

// Promise { <fullfiled>: 'bar' }
```

```
Promise.reject('bar')
  .then(
    null,
    err => Promise.resolve('bar')
  )

// Promise { <fullfiled>: 'bar' }
```

NB: Promise, возвращаемый `.then()`, копирует поведение promise, вернувшегося из коллбека

```
Promise.reject('bar')
  .then(() => 'Надеюсь, всё будет хорошо')

// Promise { <rejected>: 'bar' }
```

NB: если в then нет второго аргумента, и исходный promise падает с ошибкой, то получившийся промис тоже упадет с той же ошибкой.

```
Promise.reject('bar')
  .then(() => 'Надеюсь, всё будет хорошо')
  .then(() => 'Меня проигнорируют')
  .then(() => 'И меня тоже')
  .then(
    null,
    error => error.toUpperCase()
  )

// Promise { <fullfiled>: 'BAR' }
```

Promise.prototype.catch()

```
Promise.reject('foo')
  .then(
    null,
    () => 'bar'
  )

// Promise { <fullfiled>: 'bar' }
```

```
Promise.reject('foo')
  .catch(
    () => 'bar'
  )

// Promise { <fullfiled>: 'bar' }
```

```
Promise.reject('bar')
  .then(() => 'Надеюсь, всё будет хорошо')
  .catch(error => error.toUpperCase())

// Promise { <fullfiled>: 'BAR' }
```

```
somePromise
  .then(
    result => { ... },
    error => console.log(error)
  )
```

```
somePromise
  .then(result => { ... })
  .catch(error => console.log(error))
```

Promise.prototype.finally()

```
Promise.resolve('foo')
  .finally(() => doSomething())

// Promise { <fullfiled>: 'foo' }

Promise.reject('bar')
  .finally(() => doSomething())

// Promise { <rejected>: 'bar' }
```

```
showLoader();

doSomethingAsync()
  .then(result => processResult(result))
  .catch(error => showError(error))
  .finally(() => hideLoader());
```

```
const promise = Promise.resolve('foo');

promise.then(console.log); // 'foo'

promise.then(console.log); // 'foo'

promise.then(console.log); // 'foo'
```

Вернемся к примеру

Promise.all()

```
Promise.all([
  asyncProcessA(),
  asyncProcessB()
])
  .then(([resultA, resultB]) => {
    // сделать что-нибудь с
    // resultA и resultB (только если все они fullfiled)
  })
```

Promise.allSettled()

```
Promise.allSettled([
  asyncProcessA(),
  asyncProcessB()
])
  .then(([resultA, resultB]) => {
    // сделать что-нибудь с
    // resultA и resultB (вне зависимости от их состояния)
  })
```

Promise.race()

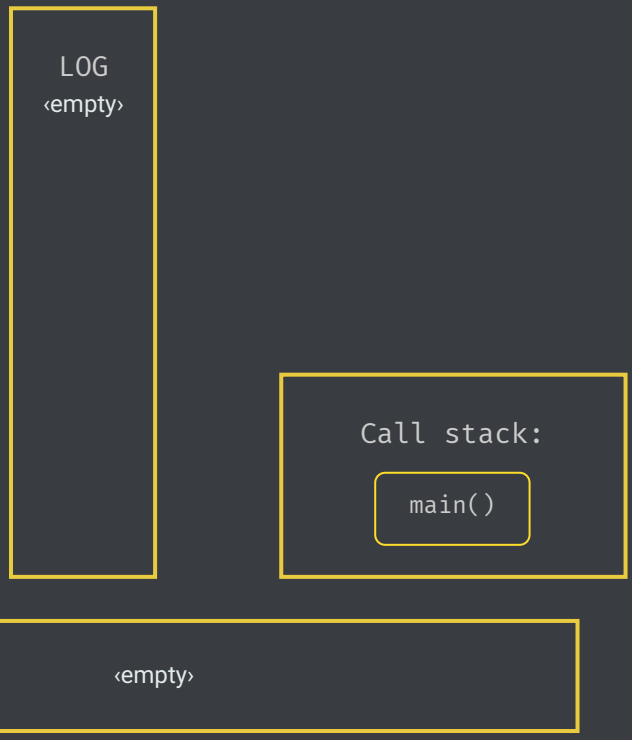
```
Promise.race([
  asyncProcessA(),
  asyncProcessB()
])
  .then(result => {
    // result - результат
    // самого быстрого из промисов
  })
```

```
console.log(1)

setTimeout(function() {
  console.log(2)
})

Promise.resolve(3).then(console.log)

console.log(4)

setTimeout(function() {
  console.log(5)
}, 0)

console.log(6)
```

```
console.log(1)

setTimeout(function() {
  console.log(2)
})

Promise.resolve(3).then(console.log)

console.log(4)

setImmediate(function() {
  console.log(5)
})

console.log(6)
```

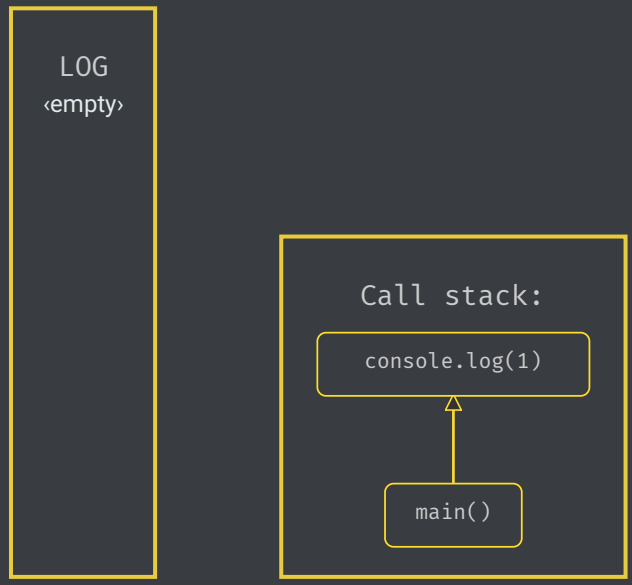
```
// 1
// 4
// 6
// 3
// 2
// 5
```

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

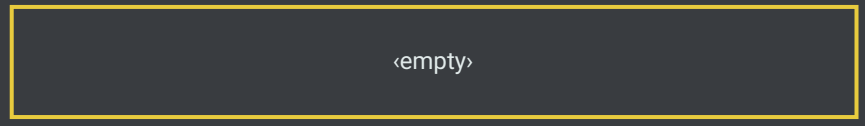


Task queue:

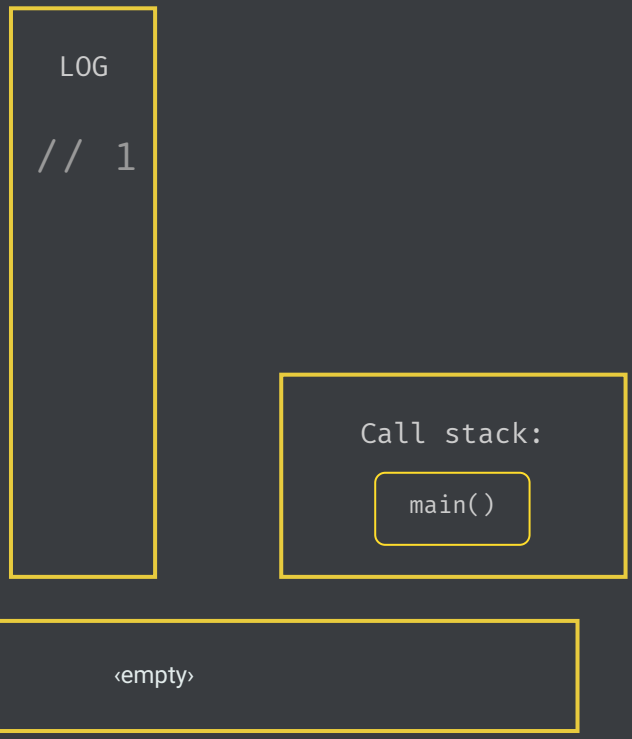
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Task queue:

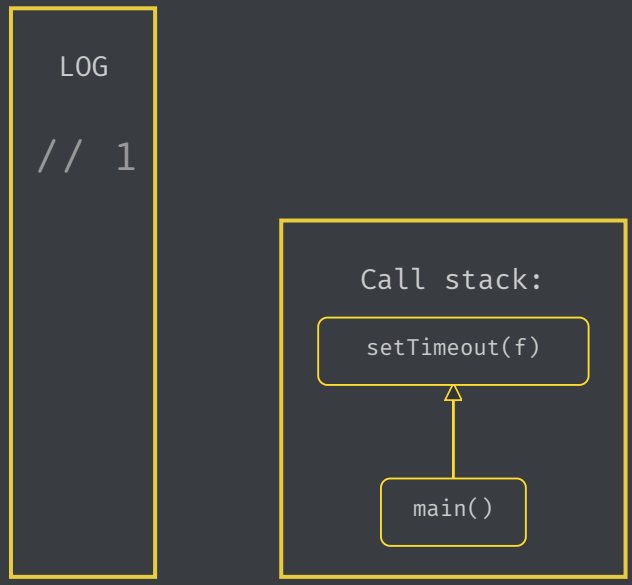


```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

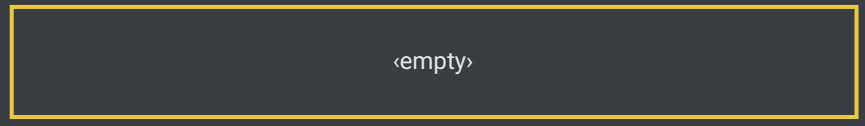


Task queue:

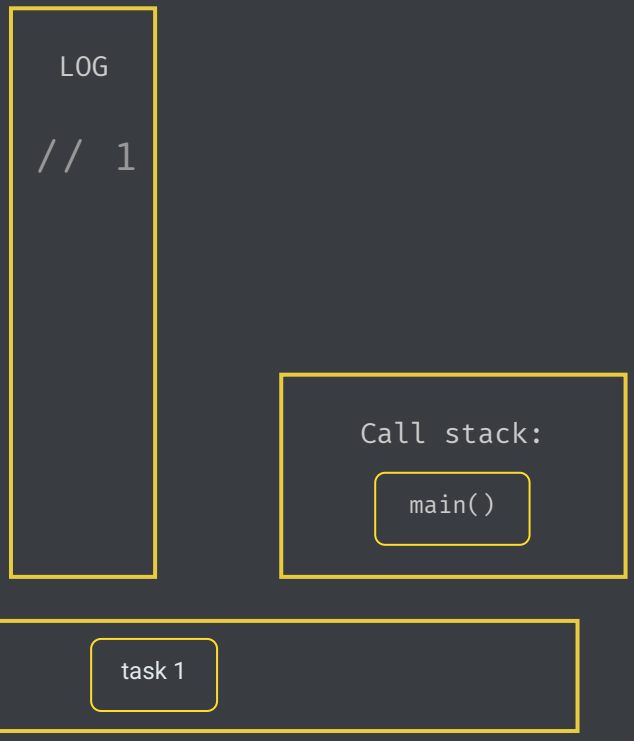
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Task queue:

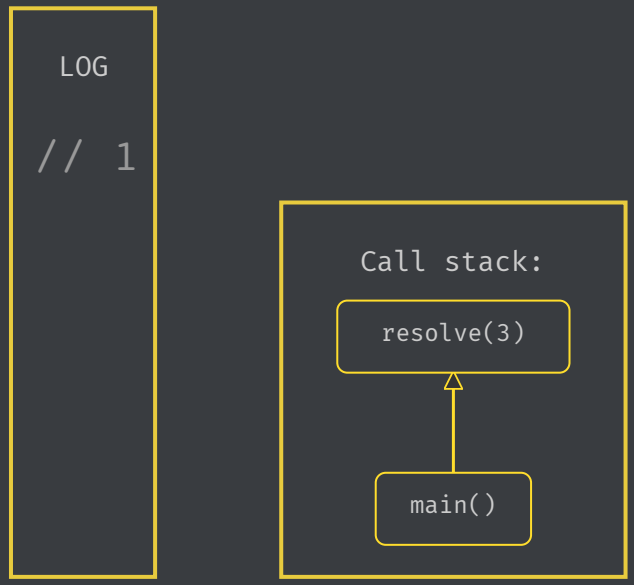


```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

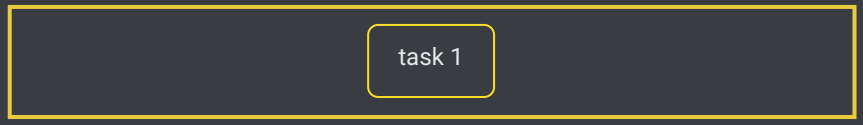


Task queue:

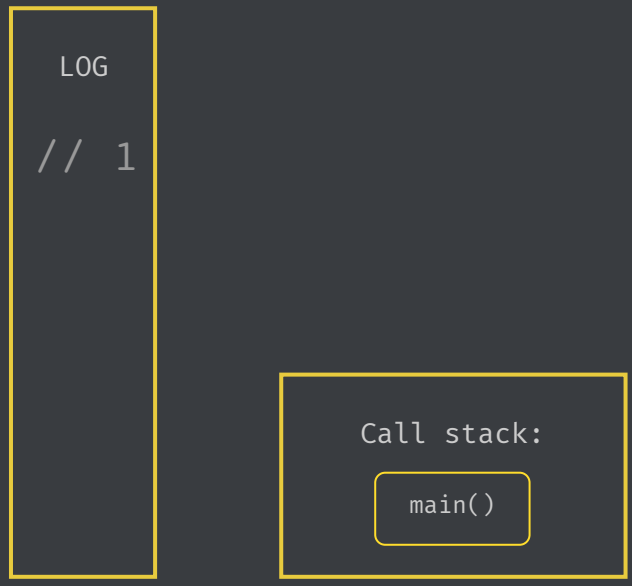
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Task queue:



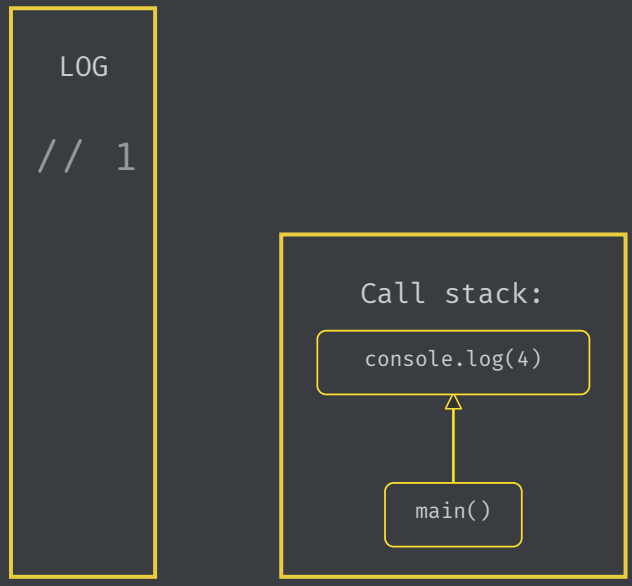
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Microtask queue:

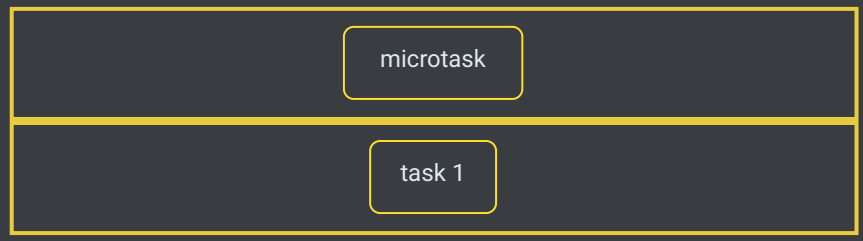
Task queue:

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

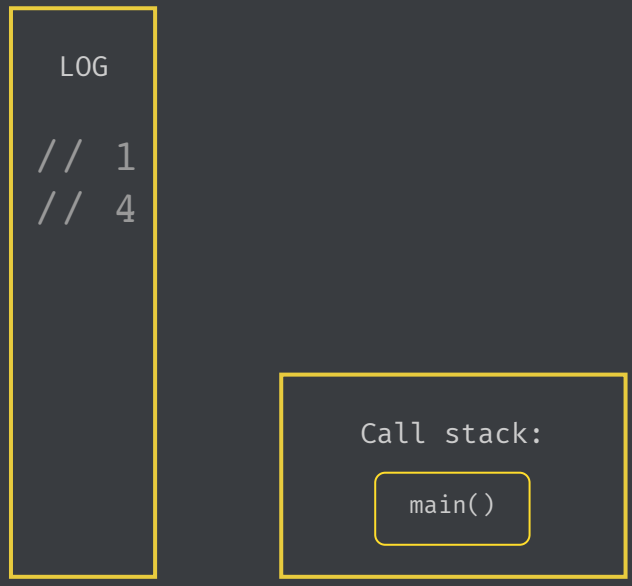


Microtask queue:

Task queue:



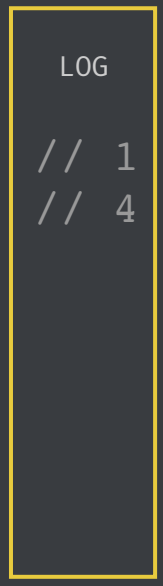
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

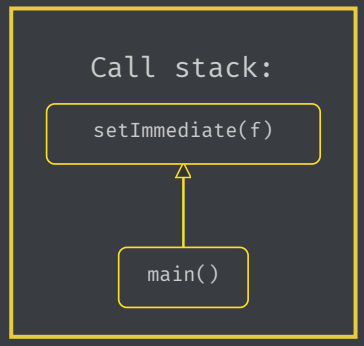


Microtask queue:

Task queue:

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

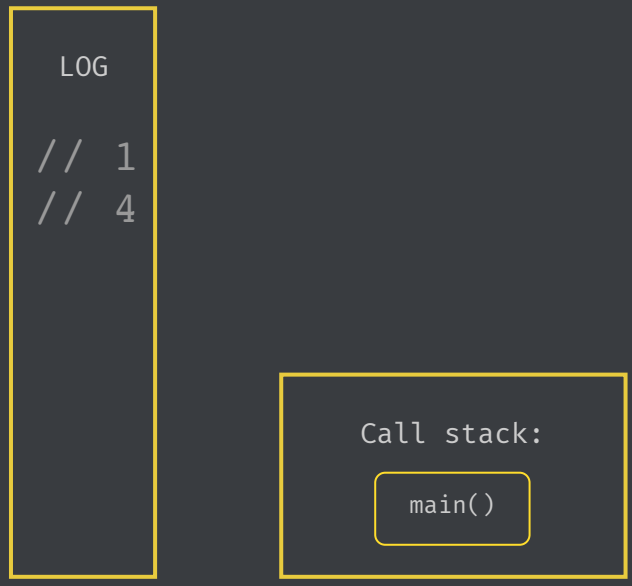




Microtask queue:

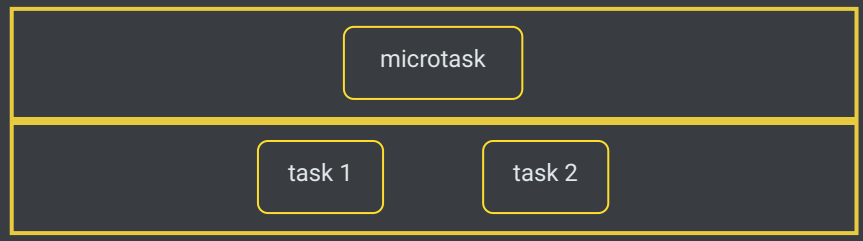
Task queue:

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

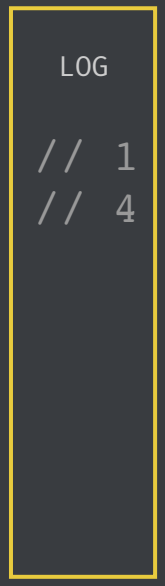


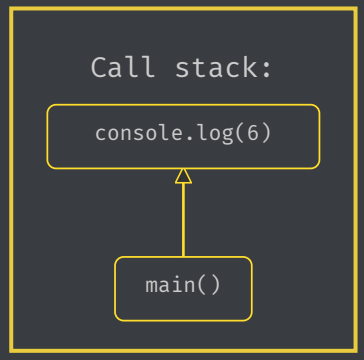
Microtask queue:

Task queue:



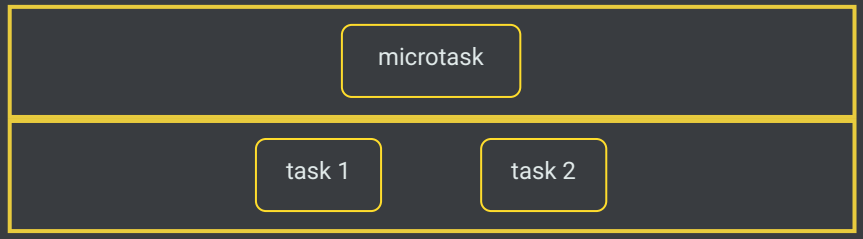
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



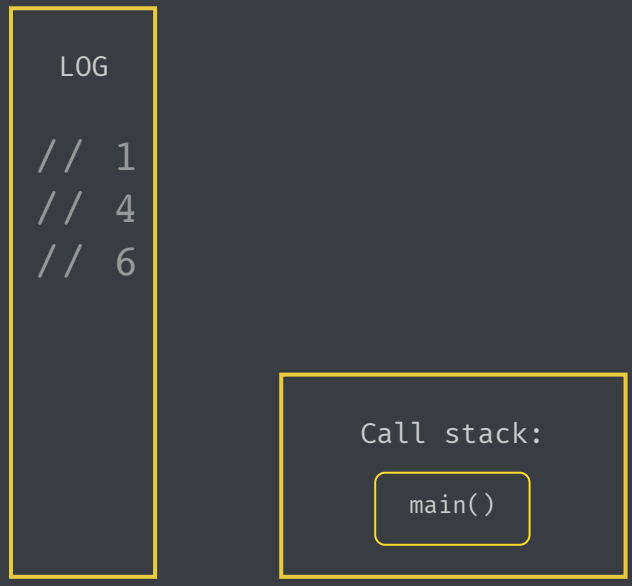


Microtask queue:

Task queue:



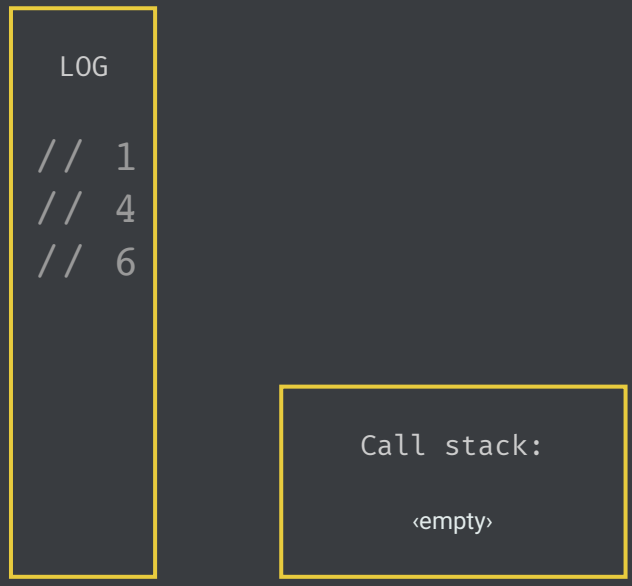
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Microtask queue:

Task queue:

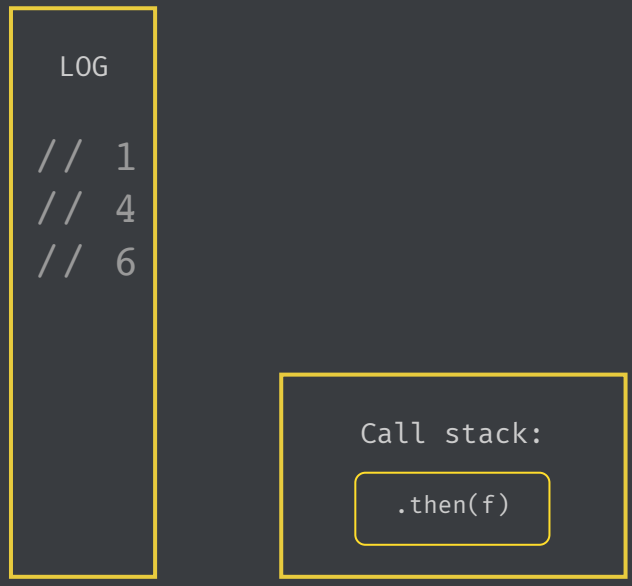
```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



Microtask queue:

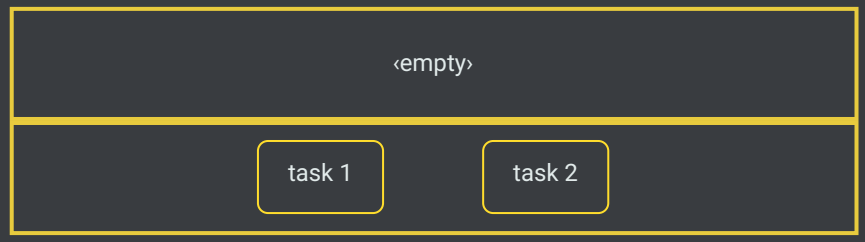
Task queue:

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

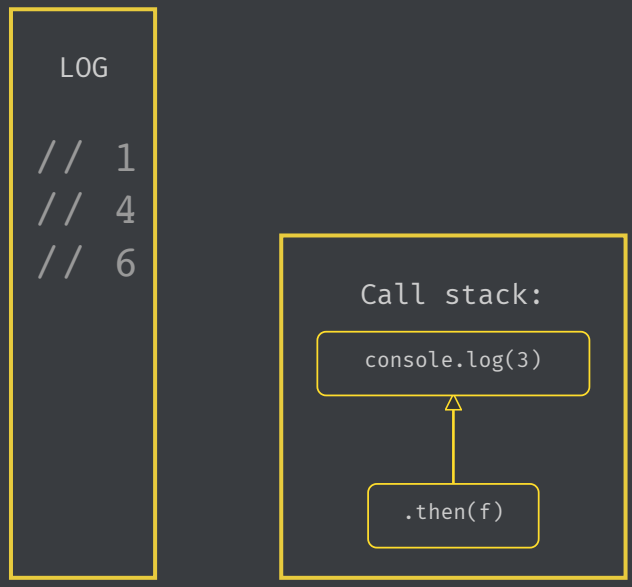


Microtask queue:

Task queue:

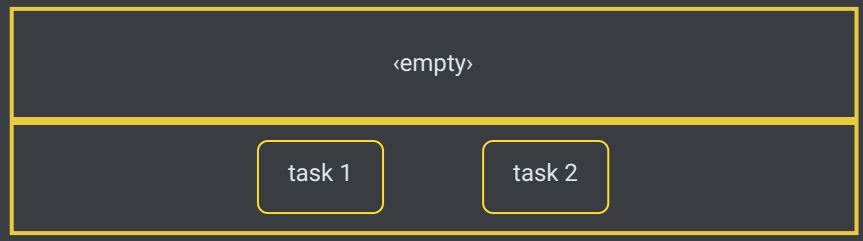


```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

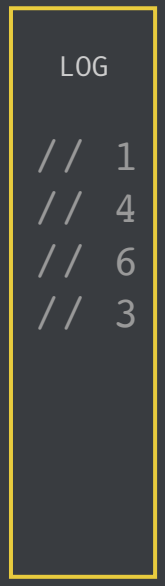


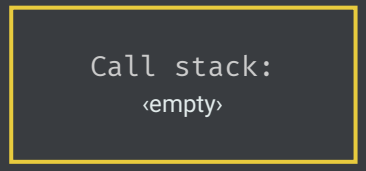
Microtask queue:

Task queue:

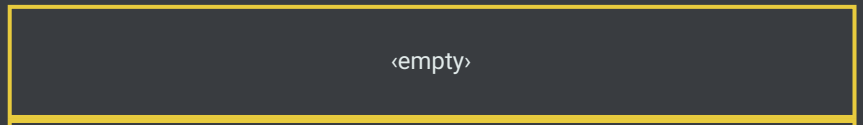


```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```



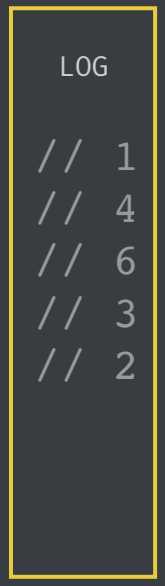


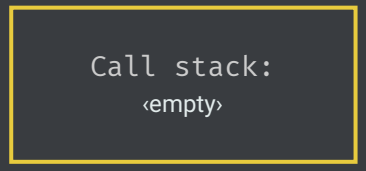
Microtask queue:



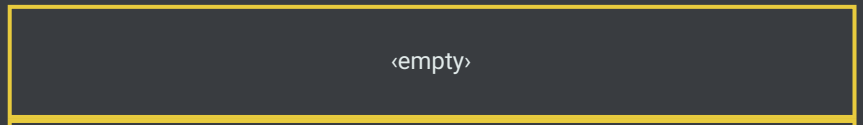
Task queue:

```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

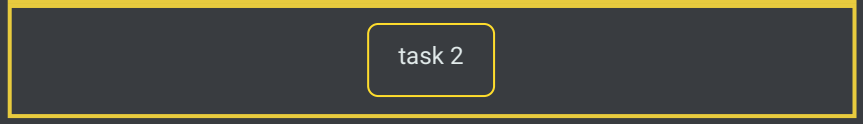




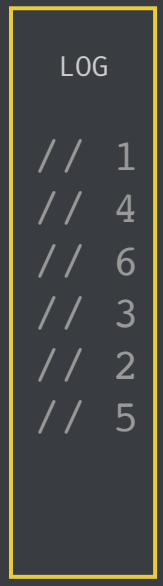
Microtask queue:

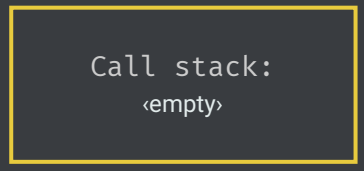


Task queue:

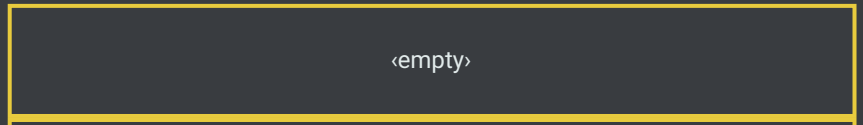


```
console.log(1);

setTimeout(() =>
  console.log(2));

Promise.resolve(3)
  .then(x => console.log(x));

console.log(4);

setImmediate(() =>
  console.log(5));

console.log(6);
```

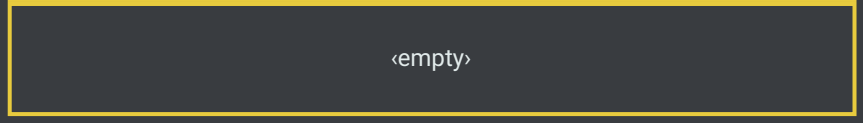




Microtask queue:



Task queue:



TASK QUEUE

1. Асинхронных задания: setTimeout, setImmediate, setInterval, XMLHttpRequest и тд
2. Отдает задачи на определенном этапе Event Loop (после синхронных операций)
3. Очередь не обрабатывает таймауты и пр, только принимает задачи по их завершении
4. Выполняется по одной задаче за цикл Event Loop

MICROTASK QUEUE

1. Асинхронные задания: Promise, fetch, async/await и тд
2. Запускается сразу после того как пустеет callstack
3. Выполняется пока очередь не опустеет, включая новые добавленные задачи

```
const f = async () => {
  return 1;
}

f().then(console.log); // 1
```

```
const f = () => {
  return Promise.resolve(1);
}

f().then(console.log); // 1
```

```
const f = async () => {
  const result1 = await someRequest();
  const result2 = await someRequest(result1);

  return someRequest(result2);
}
```

МНОГО ЗАПРОСОВ НА ПРОМИСАХ

```
const getResult = (data) => {
  return makeRequest(url1, data)
    .then(result => makeRequest(url2, result))
    .then(result => makeRequest(url3, result))
    .then(result => Promise.all([
      makeRequest(url4, result),
      makeRequest(url5, result)
    ])
    .then(([r1, r2]) => makeRequest(url6, r1, r2))
    .then(result => makeRequest(url7, result)
    .then(result => doSmthWith(result))
 }
```

МНОГО ЗАПРОСОВ НА ASYNC/AWAIT

```
const getResult = async (data) => {
  const result1 = await makeRequest(url1, data);
  const result2 = await makeRequest(url2, result1);
  const result3 = await makeRequest(url3, result2);

  const [result4, result5] = await Promise.all([
    makeRequest(url4, result3),
    makeRequest(url5, result3)
  ]);

  const result6 = makeRequest(url6, result4, result5);
  const result7 = makeRequest(url7, result6);

  return doSmthWith(result7);
}
```

ОБРАБОТКА ОШИБОК

```
const getResult = async (data) => {
  try {
    const result1 = await makeRequest(url1, data);
    const result2 = await makeRequest(url2, result1);
    const result3 = await makeRequest(url3, result2);

    const [result4, result5] = await Promise.all([
      makeRequest(url4, result3),
      makeRequest(url5, result3)
    ]);

    const result6 = makeRequest(url6, result4, result5);
    const result7 = makeRequest(url7, result6);

    return doSmthWith(result7);
  } catch (e) {
    return 'Sorry, error';
  }
}
```

СПИСОК СУПЕРВАЖНЫХ ВЕЩЕЙ, КОТОРЫЕ НУЖНО ДОМА ПРОРАБОТАТЬ И ЗАПОМНИТЬ

1. Как работает event loop

СПИСОК СУПЕРВАЖНЫХ ВЕЩЕЙ, КОТОРЫЕ НУЖНО ДОМА ПРОРАБОТАТЬ И ЗАПОМНИТЬ

1. setTimeout, setInterval, контекст вызова коллбека, как отменить таймаут
2. Promise: как создать, как управлять
3. Что происходит при ошибке в Promise
4. XHR, fetch, как работает, зачем применяется
5. requestAnimationFrame

ОБЯЗАТЕЛЬНО К ПРОСМОТРУ

- Jake Archibald: In The Loop (на русском)
- Philip Roberts: What the heck is the event loop anyway?
- Джейк Арчибальд. В цикле

```
Promise.reject('a')
    .catch(p => p + 'b')
    .catch(p => p + 'c')
    .then(p => p + 'd')
    .finally(p => p + 'e')
    .then(p => console.log(p))

console.log('f'); // ?
```

```
Promise.reject('a')
    .catch(p => p + 'b')
    .catch(p => p + 'c')
    .then(p => p + 'd')
    .finally(p => p + 'e')
    .then(p => console.log(p))

console.log('f');
// f
// abd
```

По всем вопросам пишите в телеграм:

- Общий чат
- @markitosha

- Управление классами
- Управление элементами