Web Academy

Programming Courses

Лекция 10

https://codesandbox.io/s/lesson29062020-u4qfq

Зинченко Андрей

web-academy.com.ua

План

- Планирование: setTimeout и setInterval
- Event loop
- callback
- Promise

2

Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

setTimeout позволяет вызвать функцию один раз через определённый интервал времени.
setInterval позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

Планирование: setTimeout и setInterval

setTimeout

Синтаксис:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Параметры:

`func|code`

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

`delay`

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

`arg1`, `arg2`...

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Планирование: setTimeout и setInterval

```
1  function sayHi() {
2    alert('Привет');
3  }
4
5  setTimeout(sayHi, 1000);
```

```
1  function sayHi(phrase, who) {
2    alert( phrase + ', ' + who );
3  }
4
5  setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон
```

Планирование: setTimeout и setInterval

Отмена через clearTimeout

Вызов `setTimeout` возвращает «идентификатор таймера» `timerId`, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

Планирование: setTimeout и setInterval

setInterval

Метод `setInterval` имеет такой же синтаксис как `setTimeout`:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

Все аргументы имеют такое же значение. Но отличие этого метода от `setTimeout` в том, что функция запускается не один раз, а периодически через указанный интервал времени.

Чтобы остановить дальнейшее выполнение функции, необходимо вызвать `clearInterval(timerId)`.

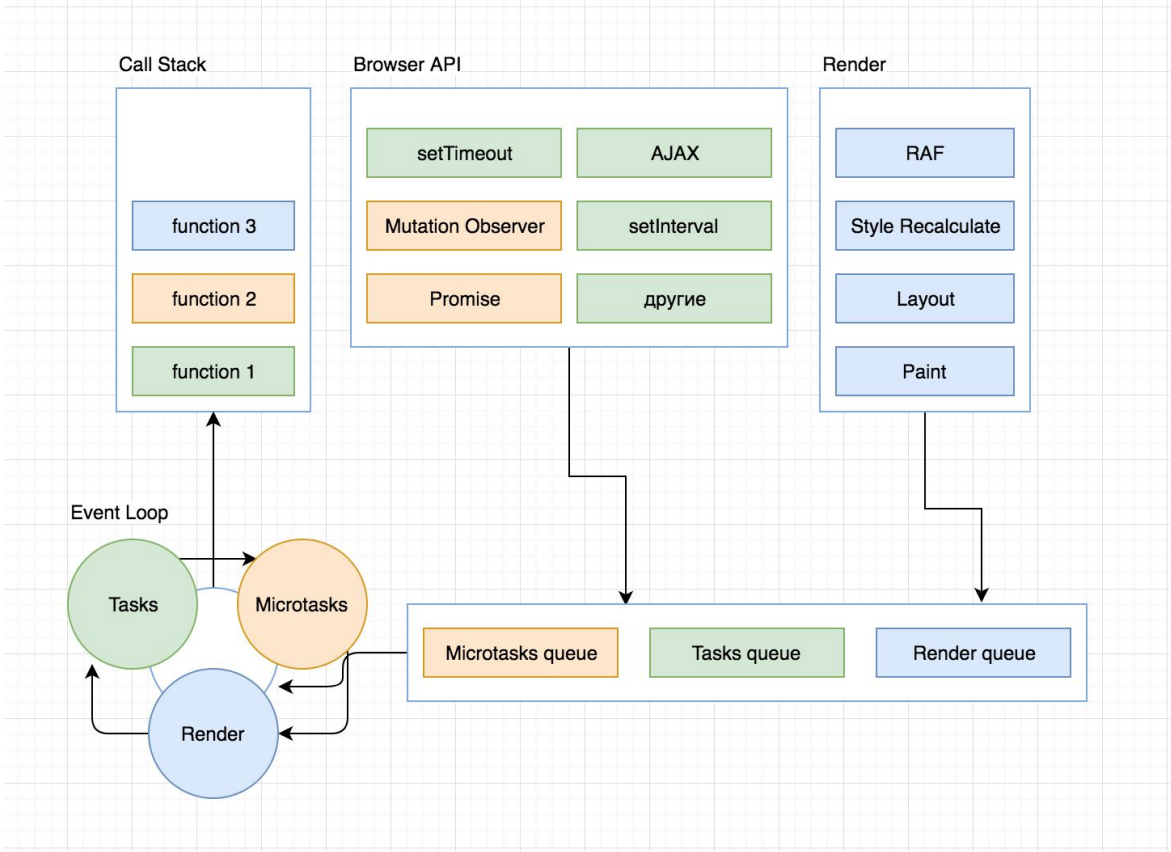
Планирование: setTimeout и setInterval

- Методы **setInterval(func, delay, ...args)** и **setTimeout(func, delay, ...args)** позволяют выполнять func регулярно или только один раз после задержки delay, заданной в мс.
- Для отмены выполнения необходимо вызвать **clearInterval/clearTimeout** со значением, которое возвращают методы **setInterval/setTimeout.**
- Планирование с нулевой задержкой **setTimeout(func,0)** или, что то же самое, **setTimeout(func)** используется для вызовов, которые должны быть исполнены как можно скорее, **после завершения исполнения текущего кода**.
- Обратим внимание, что все методы планирования **не гарантируют точную задержку**.

Event Loop

Как JavaScript может быть асинхронным и однопоточным?» Если кратко, то JavaScript однопоточный, а асинхронное поведение не является частью самого языка; вместо этого оно построено на основе него в браузере (или среде программирования) и доступно через браузерные API.

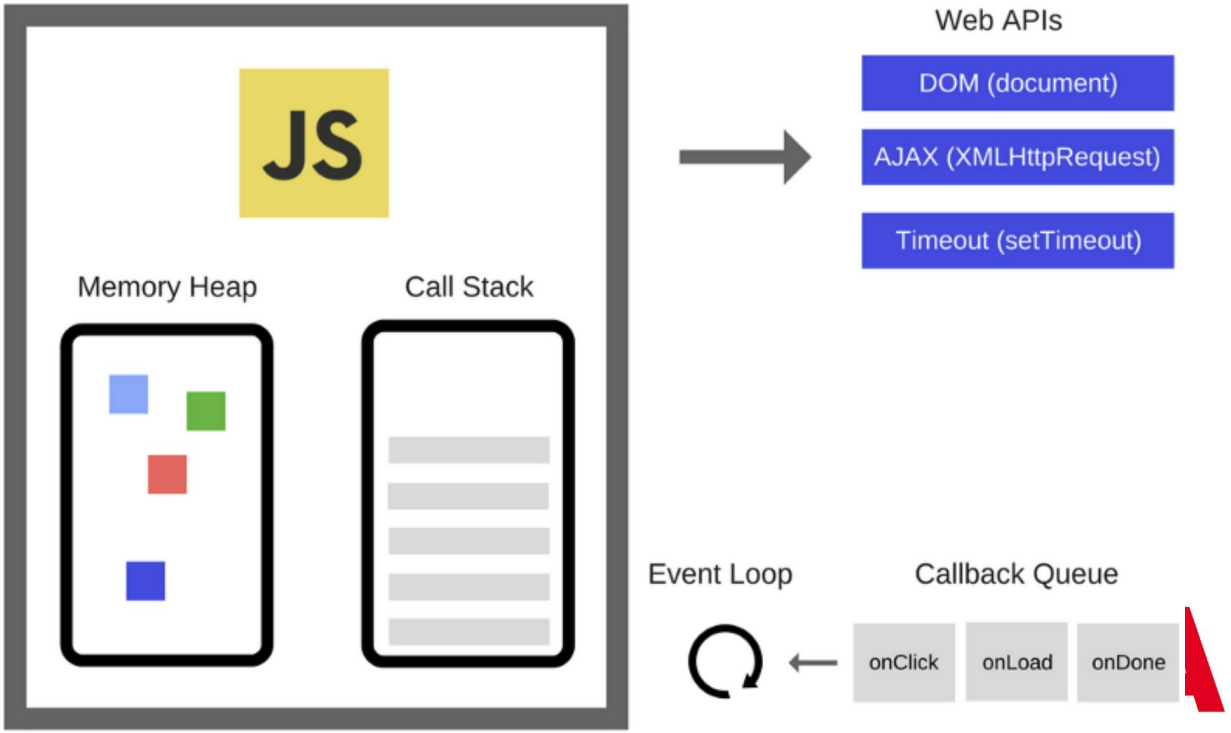
Event Loop


Call Stack
function 3
function 2
function 1
Browser API
setTimeout
AJAX
Mutation Observer
setInterval
Promise
другие
Render
RAF
Style Recalculate
Layout
Paint
Event Loop
Tasks
Microtasks
Render
Microtasks queue
Tasks queue
Render queue

Event Loop

Куча (Memory Heap) — то место, где происходит выделение памяти.

Стек вызовов (Call Stack) — то место, куда в процессе выполнения кода попадают так называемые стековые кадры.



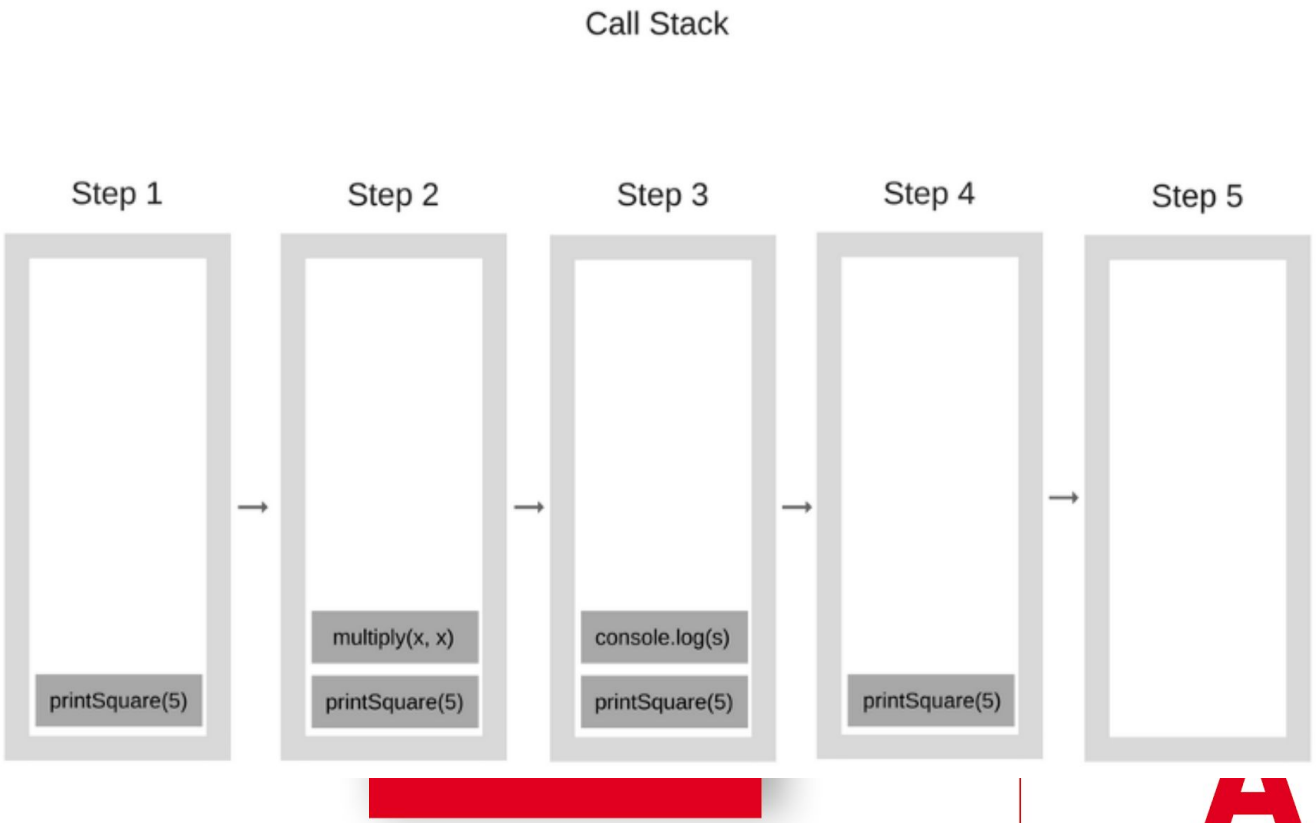
Event Loop

JavaScript — однопоточный язык программирования. Это означает, что у него один стек вызовов. Таким образом, в некий момент времени он может выполнять лишь **какую-то одну задачу**.

Стек вызовов — это структура данных, которая, говоря упрощенно, записывает сведения о месте в программе, где мы находимся. Если мы переходим в функцию, мы помещаем запись о ней в верхнюю часть стека. Когда мы из функции возвращаемся, мы вытаскиваем из стека самый верхний элемент и оказываемся там, откуда вызывали эту функцию. Это — всё, что умеет стек.

Event Loop

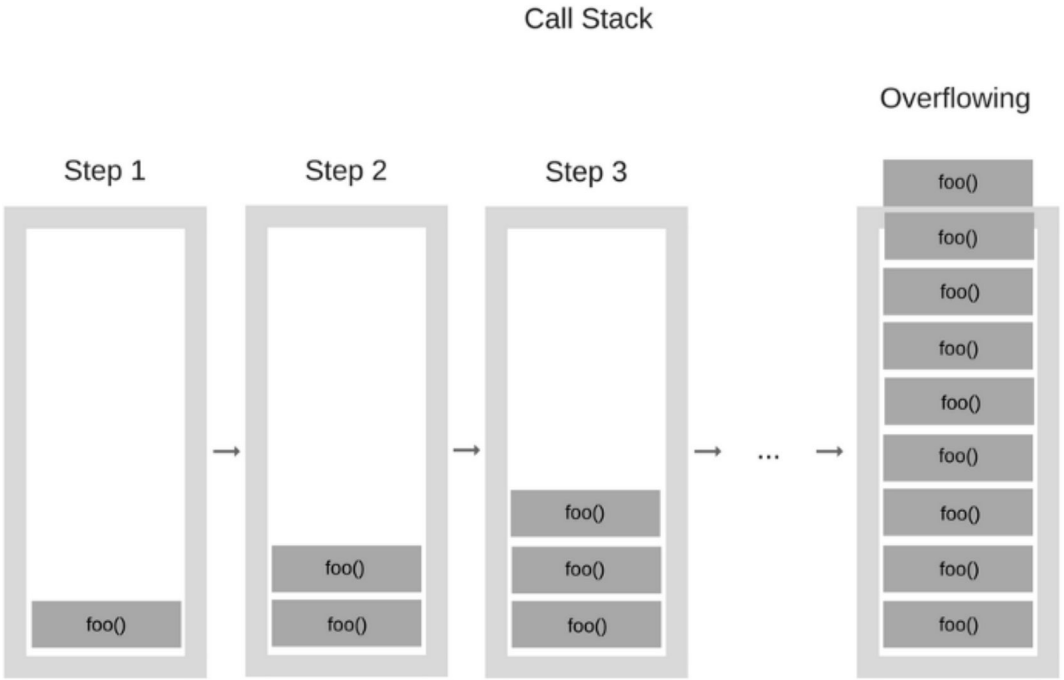
```
function multiply(x, y) {
    return x * y;
}
function printSquare(x) {
    var s = multiply(x, x);
    console.log(s);
}
printSquare(5);
```



Event Loop

```
function foo() {
    foo();
}
foo();
```

Event Loop




```
▶Uncaught RangeError: Maximum call stack size exceeded
```

Превышение максимального размера стека вызовов

Event Loop

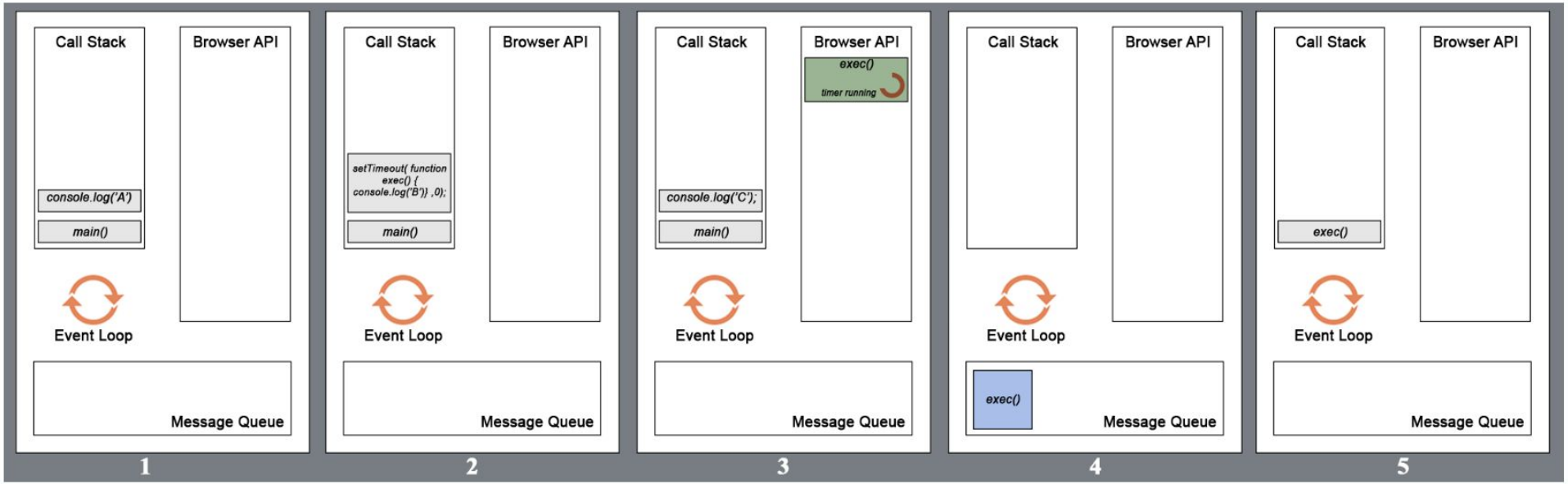
```
function main() {

console.log('A')

setTimeout(function exec() {
  console.log('B')
}, 0)

console.log('C')
}
```

Event Loop


Call Stack
Browser API
console.log('A')
main()
Event Loop
Message Queue
1
Call Stack
Browser API
setTimeout(function exec() { console.log('B')} ,0);
main()
Event Loop
Message Queue
2
Call Stack
Browser API
exec()
timer running
console.log('C');
main()
Event Loop
Message Queue
3
Call Stack
Browser API
Event Loop
exec()
Message Queue
4
Call Stack
Browser API
exec()
Event Loop
Message Queue
5

Event Loop

callback

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом.

Если после вызова `loadScript(…)` есть какой-то код, то он не будет ждать, пока скрипт загрузится.

```
loadScript('/my/script.js');
// код, написанный после вызова функции loadScript,
// не будет дожидаться полной загрузки скрипта
// ...
```

callback

Давайте передадим функцию `callback` вторым аргументом в `loadScript`, чтобы вызвать её, когда скрипт загрузится:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

callback

Теперь, если мы хотим вызвать функцию из скрипта, нужно делать это в колбэке:

```
1  loadScript('/my/script.js', function() {
2    // эта функция вызовется после того, когда загрузится скрипт
3    newFunction(); // теперь всё работает
4    ...
5  });
```

callback

Как нам загрузить два скрипта один за другим: сначала первый, а за ним второй?

callback

```
1  loadScript('/my/script.js', function(script) {
2
3    alert(`Здорово, скрипт ${script.src} загрузился, загрузим ещё один`);
4
5    loadScript('/my/script2.js', function(script) {
6      alert(`Здорово, второй скрипт загрузился`);
7    });
8
9  });
```

callback

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...и так далее, пока все скрипты не будут загружены
    });

  })

});
```

callback

Перехват ошибок

В примерах выше мы не думали об ошибках. А что если загрузить скрипт не удалось? Колбэк должен уметь реагировать на возможные проблемы.

Ниже улучшенная версия `loadScript`, которая умеет отслеживать ошибки загрузки:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

  document.head.append(script);
}
```

callback

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // обрабатываем ошибку
  } else {
    // скрипт успешно загружен
  }
});
```

callback

```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...и так далее, пока все скрипты не будут загружены (*)
          }
        });

      }
    })
  }
});
```

callback

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...и так далее, пока все скрипты не будут загружены (*)
  }
};
```

Promise

Представьте, что вы известный певец, которого фанаты постоянно донимают расспросами о предстоящем сингле.

Чтобы получить передышку, вы обещаете разослать им сингл, когда он будет выпущен. Вы даёте фанатам список, в который они могут записаться. Они могут оставить там свой e-mail, чтобы получить песню, как только она выйдет. И даже больше: если что-то пойдёт не так, например, в студии будет пожар и песню выпустить не выйдет, они также получат уведомление об этом.

Все счастливы! Вы счастливы, потому что вас больше не донимают фанаты, а фанаты могут больше не беспокоиться, что пропустят новый сингл.

Promise

Это аналогия из реальной жизни для ситуаций, с которыми мы часто сталкиваемся в программировании:

Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. В нашей аналогии это – **«певец»**.

Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции. Это – **«фанаты»**.

Promise (по англ. promise, будем называть такой объект «промис») – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. В терминах нашей аналогии – это **«список для подписки»**. «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Promise

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. В терминах нашей аналогии: исполнитель – это «певец».

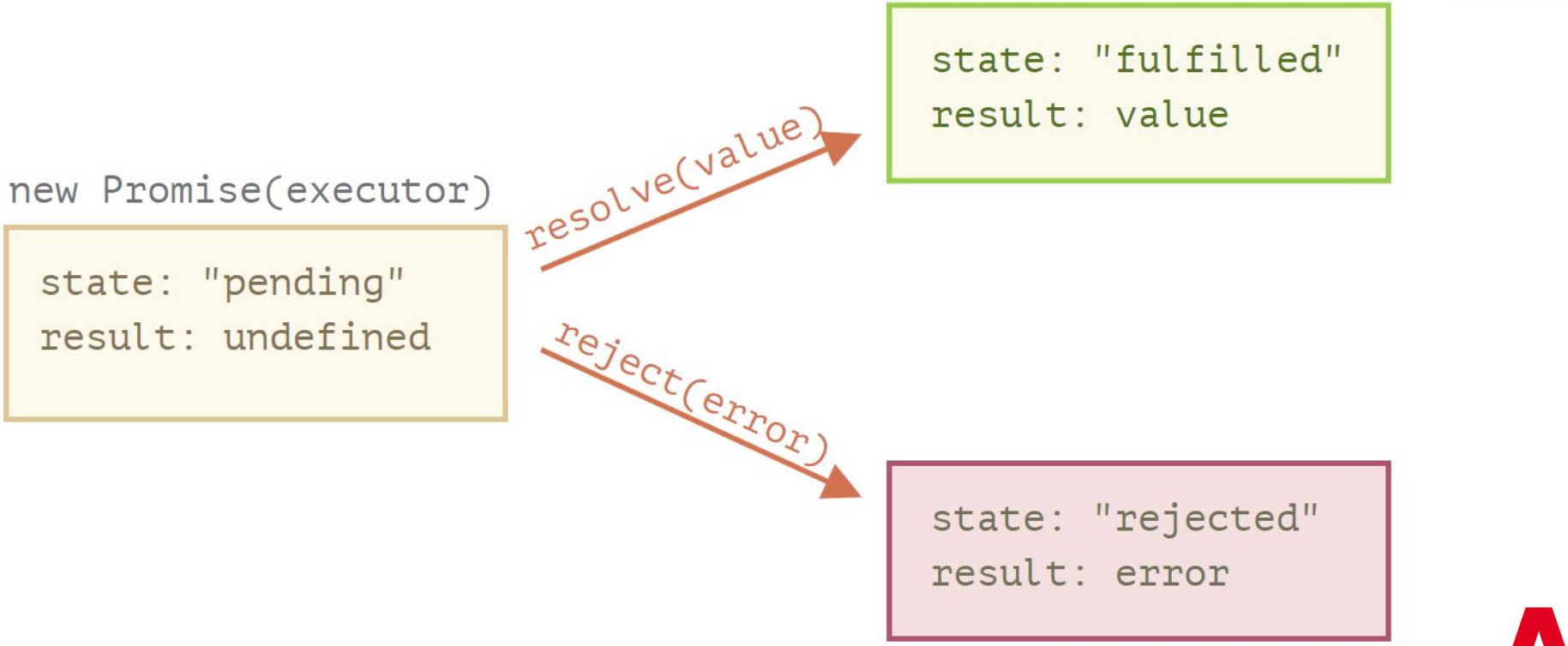
Ее аргументы resolve и reject – это колбеки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

Когда он получает результат, сейчас или позже – не важно, он должен вызвать один из этих колбэков:

resolve(value) — если работа завершилась успешно, с результатом value.
reject(error) — если произошла ошибка, error – объект ошибки.

```
1  let promise = new Promise(function(resolve, reject) {
2    // функция-исполнитель (executor)
3    // "певец"
4  });
```

Promise


new Promise(executor)
state: "pending"
result: undefined
resolve(value)
reject(error)
state: "fulfilled"
result: value
state: "rejected"
result: error

Promise

```
let promise = new Promise(function(resolve, reject) {
  // эта функция выполнится автоматически, при вызове new Promise

  // через 1 секунду сигнализировать, что задача выполнена с результатом
  setTimeout(() => resolve("done"), 1000);
});
```

Promise

Мы можем наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове `new Promise`.
2. Исполнитель получает два аргумента: `resolve` и `reject` — это функции, встроенные в JavaScript, поэтому нам не нужно их писать. Нам нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет `resolve("done")`, чтобы передать результат:

Promise

```
let promise = new Promise(function(resolve, reject) {
  // спустя одну секунду будет сообщено, что задача выполнена с ошибкой
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

Promise

Потребители: then, catch, finally

Объект `Promise` служит связующим звеном между исполнителем («создающим» кодом или «певцом») и функциями-потребителями («фанатами»), которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов `.then`, `.catch` и `.finally`.

Promise

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

then

Наиболее важный и фундаментальный метод – `.then`.

Синтаксис:

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

Promise

catch

Если мы хотели бы только обработать ошибку, то можно использовать `null` в качестве первого аргумента: `.then(null, errorHandlingFunction)`. Или можно воспользоваться методом `.catch(errorHandlingFunction)`, который сделает тоже самое:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

// .catch(f) это тоже самое, что promise.then(null, f)
promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

Вызов `.catch(f)` – это сокращённый, «укороченный» вариант `.then(null, f)`.

Promise

finally

По аналогии с блоком `finally` из обычного `try {...} catch {...}`, у промисов также есть метод `finally`.

Вызов `.finally(f)` похож на `.then(f, f)`, в том смысле, что `f` выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

`finally` хорошо подходит для очистки, например остановки индикатора загрузки, его ведь нужно остановить вне зависимости от результата.

Например:

```
new Promise((resolve, reject) => {
  /* сделать что-то, что займёт время, и после вызвать resolve/reject */
})
  // выполнится, когда промис завершится, независимо от того, успешно или нет
  .finally(() => остановить индикатор загрузки)
  .then(result => показать результат, err => показать ошибку)
```

Promise

Новой функции `loadScript` не будет нужен аргумент `callback`. Вместо этого она будет создавать и возвращать объект `Promise`, который будет переходить в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

    document.head.append(script);
  });
}
```

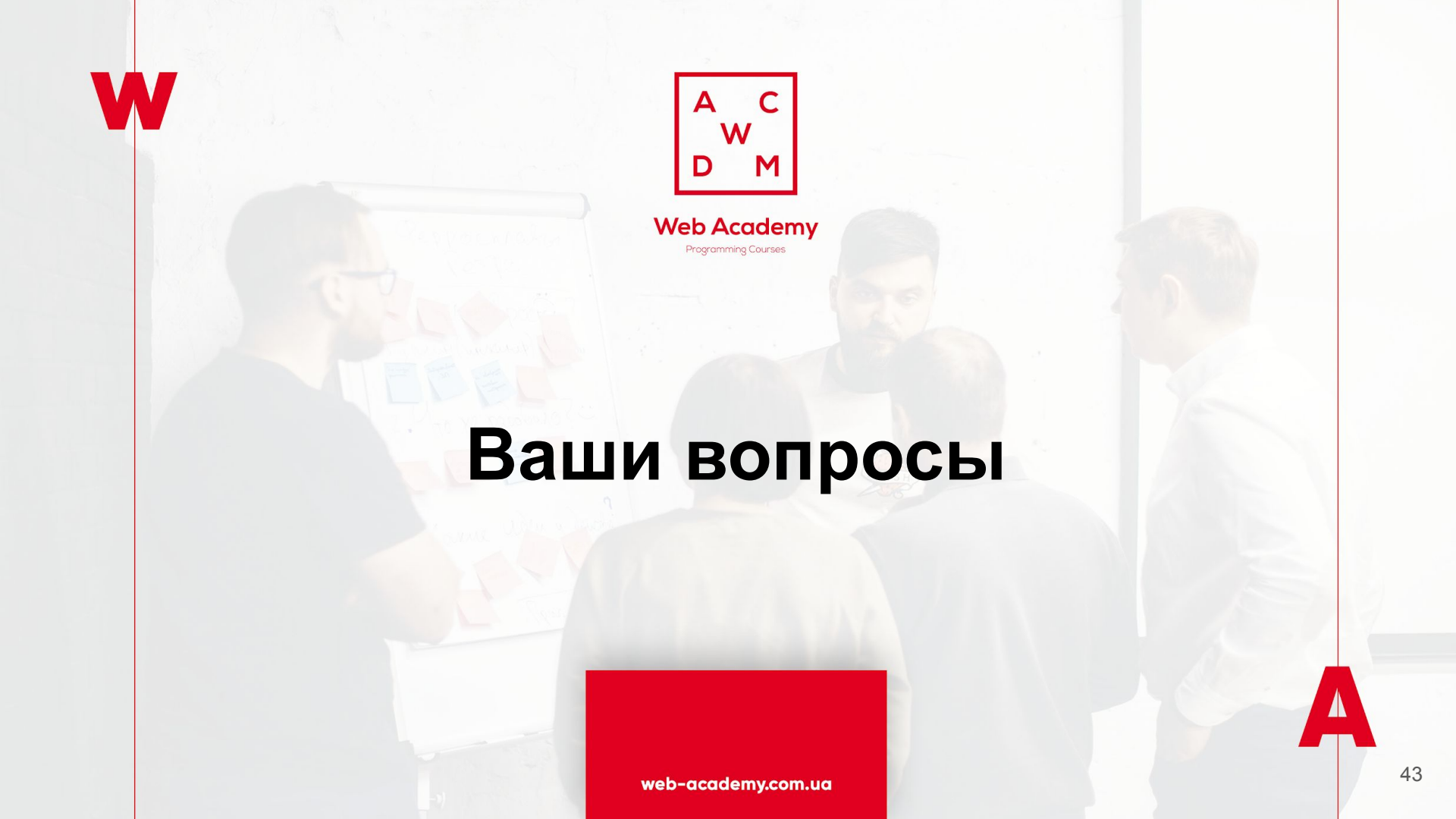
Promise

```
let promise = loadScript("https://cdnjs.cloudflare.com/

promise.then(
  script => alert(`${script.src} загружен!`),
  error => alert(`Ошибка: ${error.message}`)
);
```

Promise

Промисы	Колбэки
Промисы позволяют делать вещи в естественном порядке. Сперва мы запускаем `loadScript(script)`, и затем (`.then`) мы пишем, что делать с результатом.	У нас должна быть функция `callback` на момент вызова `loadScript(script, callback)`. Другими словами, нам нужно знать что делать с результатом *до того*, как вызовется `loadScript`.
Мы можем вызывать `.then` у `Promise` столько раз, сколько захотим. Каждый раз мы добавляем нового «фаната», новую функцию-подписчика в «список подписок». Больше об этом в следующей главе: Цепочка промисов.	Колбэк может быть только один.


Web Academy
Programming Courses
Ваши вопросы
web-academy.com.ua