

A collection of light blue geometric shapes including triangles, squares, and circles, some containing icons like a gear and a lightbulb, arranged in a scattered pattern on the left side of the slide.

АСИНХРОННОСТЬ. ПРОМИСЫ. ASYNC/AWAIT

JavaScript

Модуль 5. Урок 1.

СИНХРОННЫЙ И АСИНХРОННЫЙ КОД

синхронный код

Команды синхронного кода выполняются в том порядке, в котором они следуют в тексте программы:

```
console.log('1');  
console.log('2');  
console.log('3');
```

Результат:

1
2
3

асинхронный код

Команды асинхронного кода выполняются по мере получения результатов:

```
console.log('1');  
setTimeout(function afterTwoSeconds() {  
  console.log('2');  
}, 2000)  
console.log('3');
```

коллбэк – функция
обратного вызова

Результат:

1
3
2

*Время получения
результата часто
непредсказуемо.*

АСИНХРОННЫЙ ВЫЗОВ ФУНКЦИИ С ИСПОЛЬЗОВАНИЕМ КОЛЛБЭКА

Допустим у нас есть функция, медленно складывающая 2 числа (скажем, на сервере). После завершения операции она будет вызывать функцию-коллбэк, передавая ей результат:

```
function add(x,y,f) {  
    setTimeout(()=>f(x+y), 1000);  
}
```

Тогда, чтобы сложить 1+2, потом к результату прибавить 3, и потом его вывести, надо написать такой код:

```
add(1,2,  
    res=>add(res,3,  
        res=>console.log(res)));
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛБЭКАХ

```
arr = [1,2,3];
arr.summarize = function() {
  this.sum = 0;
  this.forEach(function(e) { this.sum = this.sum+e; } );
  // В коллбэке this указывает на window, а не на объект arr
}
console.log(arr.sum); // 0
console.log(sum); // 6 – потому что this.sum – это window.sum
```

обход проблемы (устаревший подход):

```
arr.summarize = function() {
  this.sum = 0;
  var self = this;
  this.forEach(function(e) { self.sum = self.sum+e; } ); // self попадает в замыкание и продолжает указывать на this
}
```

более современный подход – используем bind:

```
this.forEach(function(e) { this.sum = this.sum+e; }.bind(this) );
// за счет bind мы привязываем функцию к this – теперь this указывает на arr
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛЕБАХ

Еще одно решение – использовать стрелочную функцию: в ней `this` продолжает указывать на `arr`:

```
arr.summarize = function() {  
  this.sum = 0;  
  this.forEach(e=>{ this.sum = this.sum+e; });  
}
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛЛЕБАХ


Рассмотрим пример – вывод кастомизированного сообщения по клику:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // напечатает undefined: this указывает на window: метод – обычная функция  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
new Messenger("hello from Messenger").addClickHandler();
```

ИСПОЛЬЗОВАНИЕ THIS В КОЛБЭКАХ

Решение 1: используем bind(this):

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick.bind(this);  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```




при передаче ссылки
привязываем метод к this с
помощью bind()

ИСПОЛЬЗОВАНИЕ THIS В КОЛБЭКАХ

Решение 2: используем стрелочную функцию при привязке:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = ()=>this.handleClick();  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```

оборачиваем вызов в
стрелочную функцию – this
не теряется



ИСПОЛЬЗОВАНИЕ THIS В КОЛБЭКАХ

Решение 3: используем свойство вместо метода:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
    this.handleClick = () => console.log(this.message);  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
  
new Messenger("hello from Messenger").addClickHandler();
```


метода нет – объявляем свойство
как ссылку на стрелочную
функцию – тогда проблемы
потери this нет

ИСПОЛЬЗОВАНИЕ THIS В КОЛБЭКАХ

Решение 4: используем переприсваивание свойства:

```
class Messenger {  
  constructor(message) {  
    this.message = message;  
    this.handleClick = this.handleClick.bind(this);  
  }  
  
  handleClick () {  
    console.log(this.message); // hello from Messenger  
  }  
  
  addClickHandler() {  
    window.onclick = this.handleClick;  
  }  
}  
new Messenger("hello from Messenger").addClickHandler();
```

подменяем ссылку handleClick
на ссылку, привязанную к this



ПРОМИСЫ

В ES2015 появился объект, который содержит будущий результат – Promise – обещание.

Вместо вызова коллбэка мы возвращаем промис.

На момент получения в нем еще нет результата.

Но мы можем дождаться результата, используя функцию then и передав в нее обработчик результата.

```
function add(x,y) {  
  return new Promise(function(resolve) {  
    setTimeout( ()=> resolve(x+y), 1000);  
  });  
}
```

В результате код, использующий промисы вместо коллбэков, выглядит более аккуратно:

```
add(1,2)  
  .then(x=>add(-5,x))  
  .then(res=>console.log(`result = ${res}`))
```

ПРОМИСЫ: ОБРАБОТКА ИСКЛЮЧЕНИЙ

Возможны ситуации, при которой асинхронная функция отрабатывает некорректно.

Например, если это обращение к серверу – возможно сервер не отвечает или отвечает с ошибкой.

Как быть в такой ситуации? Что вернуть?

На этот случай в Promise есть второй аргумент: `reject` - отказ.

Вызывая его, мы говорим: что-то пошло не так, клиент должен обработать эту ситуацию – и вызываем `reject`, передавая туда подробности произошедшего (можно передать любой объект).

```
function add(x,y) {  
  return new Promise (function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Клиент, получая промис, может определить, как обработать ошибочную ситуацию, в методе `catch()`:

```
add(1,2) // здесь пока все нормально  
  .then(x=>add(-5,x)) // здесь что-то пошло не так – дальше не идем, прыгаем в catch  
  .then(res=>console.log(`result = ${res}`))  
  .catch(err=>console.log("ERROR:"+err)); // выводим сообщение об ошибке
```

PROMISE.ALL

Бывает ситуация, когда надо запустить на выполнение сразу несколько операций.

Например, нужно скачать с сервера сразу 3 документа. Тогда лучше их запускать всех сразу, а не последовательно. Для этого можно использовать Promise.all, передавая в него список промисов:

```
Promise.all([add(1,2),add(2,3),add(5,5)])  
  .then(res=>console.log(res))
```

> [3, 5, 10] (выведется через секунду)

СИНТАКСИС ASYNC/AWAIT

В ES2018 появился новый синтаксис – `async/await`.

Он позволяет еще лаконичнее работать с асинхронным кодом, как будто это синхронный код.

```
function add(x,y) {  
  return new Promise(function(resolve) {  
    setTimeout(()=>resolve(x+y), 1000);  
  });  
}  
  
async function main() {  
  var res = await add(1, 2);  
  var res2 = await add (res, 3);  
  console.log( res2 ); //6  
}  
  
main();
```

СИНТАКСИС ASYNC/AWAIT: ОБРАБОТКА ИСКЛЮЧЕНИЙ

В ES2018 появился новый синтаксис – `async/await`.

Он позволяет еще лаконичнее работать с асинхронным кодом, как будто это синхронный код.

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}  
  
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}  
  
main();
```

Важное ограничение:

await может использоваться только внутри **async** функции!

СИНТАКСИС ASYNC/AWAIT: ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать `async` или
`await`

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

`await` может использоваться только внутри
`async` функции!


```
main();  
console.log("FINISHED");
```

В данном случае мы не ожидаем результата работы.
FINISHED будет выведено ДО результата вычисления – команда
`main()` запускает асинхронную функцию, но не ждет ее
завершения!

СИНТАКСИС ASYNC/AWAIT: ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать `async` или
`await`



```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

`await` может использоваться только внутри
`async` функции!

```
await main();  
console.log("FINISHED");
```

Может быть так?

1



2



СИНТАКСИС ASYNC/AWAIT: ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать `async` или
`await`

```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

`await` может использоваться только внутри `async` функции!

```
await main();  
console.log("FINISHED");
```

Так нельзя! Если внешняя функция не `async`.


1

2

СИНТАКСИС ASYNC/AWAIT: ОГРАНИЧЕНИЯ

```
function add(x,y) {  
  return new Promise(function(resolve,reject) {  
    setTimeout(()=>x>0?resolve(x+y):reject("x should be >0"), 1000);  
  });  
}
```

Вы не можете здесь
использовать `async` или
`await`



```
async function main() {  
  try {  
    var res = await add(1, 2);  
    var res2 = await add (res, 3);  
    console.log( res2 ); //6  
  } catch(e) {  
    console.log("ERROR:"+e);  
  }  
}
```

Важное ограничение:

`await` может использоваться только внутри
`async` функции!


```
main().then(()=>  
  console.log("FINISHED"));
```

А вот так можно.

`main()` возвращает `Promise`.

Поэтому можно использовать `then()` и `catch()` в синхронном коде.

`FINISHED` будет напечатан ПОСЛЕ вывода результата.



ASYNC/AWAIT

// printDelayed is a Promise

```
async function printDelayed(elements) {  
  for (const element of elements) {  
    await delay(200);  
    console.log(element);  
  }  
}
```

```
async function delay(milliseconds) {  
  return new Promise(resolve => {  
    setTimeout(resolve, milliseconds);  
  });  
}
```

```
printDelayed(["Hello", "beautiful", "asynchronous", "world"]).then(() => {  
  console.log();  
  console.log("Printed every element!");  
});
```