

ПРОГРАММИРОВАНИЕ В INTERNET

АСИНХРОННОЕ ПРОГРАММИРОВАНИЕ

Синхронное =
программирование

последовательное выполнение инструкций с синхронными системными вызовами, которые полностью блокируют поток выполнения, пока системная операция (например, чтение с диска) не завершится.

JavaScript – это синхронный однопоточный ЯЗЫК

```
function a() { console.log( 'result of a()' ); }  
  
function b() { console.log( 'result of b()' ); }  
  
function c() { console.log( 'result of c()' ); }  
  
a();  
console.log('a() is done!');  
  
b();  
console.log('b() is done!');  
  
c();  
console.log('c() is done!');
```

```
PS D:\Материалы\свр_03> node .\03-01.js  
result of a()  
a() is done!  
result of b()  
b() is done!  
result of c()  
c() is done!
```

Асинхронное =
программирование

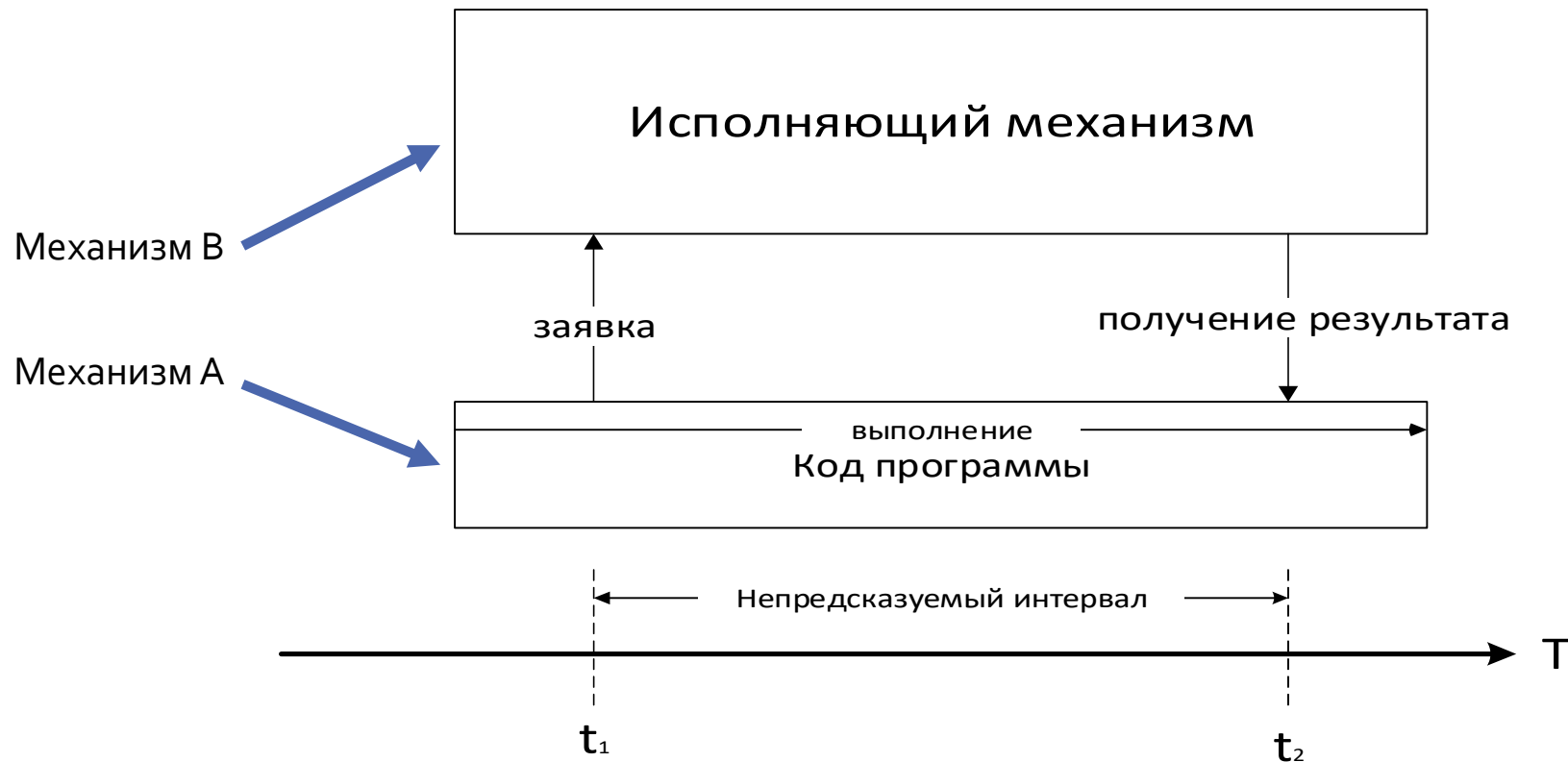
выполнение процесса в
неблокирующем режиме
системного вызова, что
позволяет потоку программы
продолжить обработку.

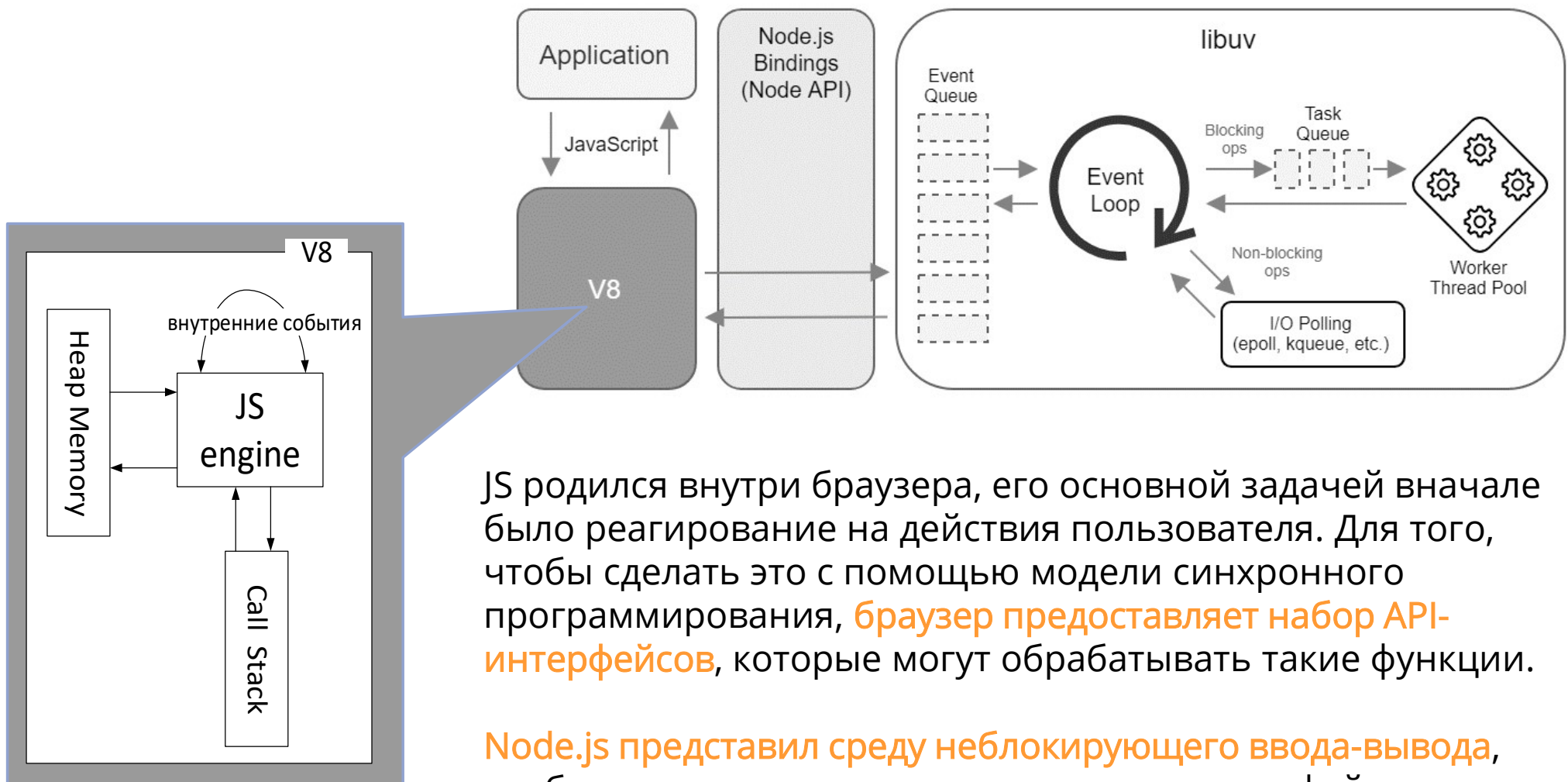
Асинхронность =

операция называется асинхронной, если ее выполнение осуществляется в 2 фазы:

- 1) **заявка на исполнение;**
- 2) **получение результата;** при этом участвуют два механизма: А-механизм, формирующий заявку и потом получающий результат; В-механизм, получающий заявку от А, исполняющий операцию и отправляющий результат А; продолжительность исполнения операции В-механизмом, как правило, непредсказуемо; в то время пока В-механизм исполняет операцию, А-механизм выполняет собственную работу.

Асинхронность в программировании





JS родился внутри браузера, его основной задачей вначале было реагирование на действия пользователя. Для того, чтобы сделать это с помощью модели синхронного программирования, **браузер предоставляет набор API-интерфейсов**, которые могут обрабатывать такие функции.

Node.js представил среду неблокирующего ввода-вывода, чтобы расширить эту концепцию на доступ к файлам, сетевые вызовы и так далее.

```

function a() {
  setTimeout( function() {
    console.log( 'result of a()' );
  }, 1000 );
}

function b() {
  setTimeout( function() {
    console.log( 'result of b()' );
  }, 500 );
}

function c() {
  setTimeout( function() {
    console.log( 'result of c()' );
  }, 1200 );
}

a();
console.log('a() is done!');

b();
console.log('b() is done!');

c();
console.log('c() is done!');

```

```

PS D:\Материалы\свр_03> node 03-02.js
a() is done!
b() is done!
c() is done!
result of b()
result of a()
result of c()

```

Функция **setTimeout(callback, delay)** – глобальная функция, которая принимает функцию обратного вызова и временно сохраняет ее. По истечении времени, заданного в миллисекундах, функция обратного вызова помещается в **очередь коллбэков (callback queue)**. Затем event loop перемещает эту функцию в **стек вызовов (call stack)**, когда стек пуст. После этого осуществляется выполнение функции обратного вызова.

В основном так работают все асинхронные операции.

Как сделать так, чтобы результат был таким?

```
result of b()  
b() is done!  
result of a()  
a() is done!  
result of c()  
c() is done!
```

Асинхронное программирование в JS

- **Callbacks**
- **Promises** (>callbacks)
- **Async/await** (> promises > callbacks)
- Async.js (> callbacks)
- Generators/yield
- Events (>observable > callbacks)
- Functor + chaining + composition
- For await + Symbol.asyncIterator

Callback
(функция обратного =
вызова)

функция, которая передается в качестве параметра другой функции и которая **будет вызвана** асинхронно обработчиком событий **после завершения задачи**.

Callbacks

Callback должен идти последним параметром в функцию.

```
function a(callback) {  
  setTimeout( () => {  
    console.log( 'result of a()' );  
    callback();  
  }, 1000 );  
}  
  
function b(callback) {  
  setTimeout( () => {  
    console.log( 'result of b()' );  
    callback();  
  }, 500 );  
}  
  
function c(callback) {  
  setTimeout( () => {  
    console.log( 'result of c()' );  
    callback();  
  }, 1200 );  
}  
  
a( () => console.log('a() is done!') );  
b( () => console.log('b() is done!') );  
c( () => console.log('c() is done!') );
```

```
PS D:\Материалы\свр_03> node 03-06.js  
result of b()  
b() is done!  
result of a()  
a() is done!  
result of c()  
c() is done!
```

Node.js callback style

Первым параметром в любой функции обратного вызова является **объект ошибки**.

Если ошибки нет, объект равен нулю. Если есть ошибка, он содержит некоторое описание ошибки и другую информацию.

Вторым, третьим, четвертым и др. идут данные, результат.

```
function callback(error, data) {  
  if (error) {  
    // обрабатываем ошибку  
  } else {  
    // обрабатываем данные  
  }  
}
```

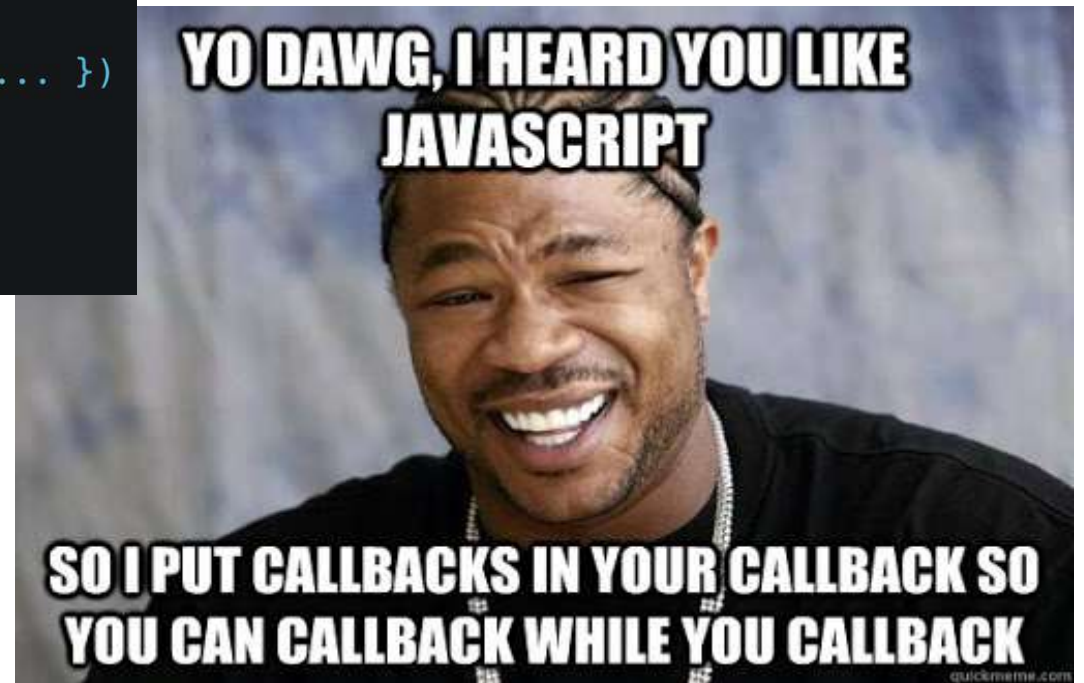
```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    // обрабатываем ошибку  
    console.log(err);  
    return;  
  }  
  // ошибок нет, обрабатываем данные  
  console.log(data);  
})
```

Callbacks

- + повторное использование кода;
- + изменение функциональности без изменения метода;
- + создание цепочек вызовов;
- падение производительности;
- ухудшение читаемости.

Callback hell, pyramid of doom

```
function printAll(){  
  printString("A", () => {  
    printString("B", () => {  
      printString("C", () => {  
        printString("D", () => { ... })  
      })  
    })  
  })  
}  
printAll();
```



Решение callback hell

```
fs.readdir(dir, (err, files) => {
  if (err) {
    // ...
  }

  files.forEach((name) => {
    if (!isImage(name)) return;

    fs.readFile(name, (err, image) => {
      if (err) {
        // ...
      }

      compress(image, (err, comp) => {
        fs.writeFile(name, comp, (err) => {
          if (err) {
            // ...
          }
          else {
            // ...
          }
        });
      });
    });
  });
});
```

1. Разбивать код на функции



```
function processFiles(err, files) {
  if (err) {
    // ...
  }
  files.forEach(checkFile);
}

function checkFile(name) {
  if (!isImage(name)) return;
  fs.readFile(name, compressImage(name));
}

function compressImage(name) {
  return (err, image) => {
    if (err) {
      // ...
    }
    compress(file, rewriteImage(name));
  }
}

function rewriteImage(name) {
  return (err, image) => {
    fs.writeFile(name, image, finishCompress);
  }
}

function finishCompress(err) {
  if (err) {
    // ...
  }
  else {
    // ...
  }
}

fs.readdir(dir, processFiles);
```


Решение callback hell

```
fs.readdir(dir, (err, files) => {
  if (err) {
    // ...
  }

  files.forEach((name) => {
    if (!isImage(name)) return;

    fs.readFile(name, (err, image) => {
      if (err) {
        // ...
      }

      compress(image, (err, comp) => {
        fs.writeFile(name, comp, (err) => {
          if (err) {
            // ...
          } else {
            // ...
          }
        });
      });
    });
  });
});
```

2. Разбитие на модули



```
function processFiles(err, files) {
  if (err) { }
  files.forEach(checkFile);
}

function checkFile(name) {
  if (!isImage(name)) return;
  fs.readFile(name, compressImage(name));
}

function compressImage(name) {
  return (err, image) => {
    if (err) { }
    compress(file, rewriteImage(name));
  }
}

function rewriteImage(name) {
  return (err, image) => {
    fs.writeFile(name, image, finishCompress);
  }
}

function finishCompress(err) {
  if (err) { }
  else { }
}

function readDir(dir) {
  fs.readdir(dir, processFiles);
}

module.export = readDir;
```

```
const compressImagesDir = require('./compressImagesDir');
compressImagesDir('/tmp/images');
```

Promise (Обещание)

ES6 / ES2015

=

объект, используемый для выполнения **отложенных и асинхронных вычислений**.

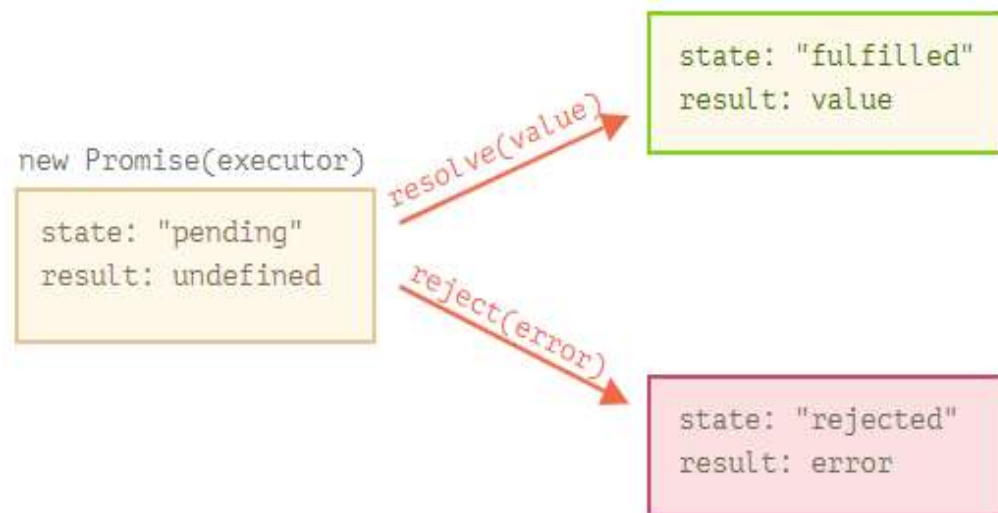
Представляет собой операцию, которая еще не завершена, но ожидается в будущем.

Promises

Свойство **state**:

- **pending** (ожидание),
- **fulfilled** (выполнено) при вызове `resolve`,
- **rejected** (отклонено) при вызове `reject`.

Свойство **result**: вначале `undefined`, далее изменяется на `value` при вызове `resolve(value)` или на `error` при вызове `reject(error)`.



Promises

resolve и **reject** – это статические методы класса Promise, которые выполняют и отклоняют промис соответственно.

Вызов `resolve =>`
обработчик then

Вызов `reject =>`
обработчик catch

```
var myPromise = new Promise( (resolve, reject) => {  
  // все прошло успешно  
  resolve('successPayload');  
  
  // что-то пошло не так  
  // reject('errorPayload');  
} );  
  
myPromise  
  .then((result) => {  
    // обрабатываем результат  
  })  
  .catch((err) => {  
    // обрабатываем ошибку  
  })  
  .finally(finallyCallback);  
  
myPromise.then((files) => { }, (err) => { }));
```

Resolve => then => finally

```
const promiseA = new Promise((resolve, reject) => {  
  setTimeout( () => {  
    resolve('result of a()');  
  }, 1000 );  
});  
  
promiseA  
  .then((result) => {  
    console.log('promiseA success:', result);  
  })  
  .catch((error) => {  
    console.log('promiseA error:', error);  
  })  
  .finally(() => {  
    console.log('a() is done!');  
  });
```

```
PS D:\Материалы\свр_03> node .\03-08.js  
promiseA success: result of a()  
a() is done!  
PS D:\Материалы\свр_03> █
```

Reject => catch => finally

```
const promiseA = new Promise((resolve, reject) => {  
  setTimeout( () => {  
    //resolve('result of a()');  
    reject('something bad happened a()');  
  }, 1000 );  
});  
  
promiseA  
  .then((result) => {  
    console.log('promiseA success:', result);  
  })  
  .catch((error) => {  
    console.log('promiseA error:', error);  
  })  
  .finally(() => {  
    console.log('a() is done!');  
  });
```

```
PS D:\Материалы\свр_03> node .\03-08.js  
promiseA error: something bad happened a()  
a() is done!  
PS D:\Материалы\свр_03> █
```


Обработка отклоненного промиса без catch

```
const promiseA = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    reject('something bad happened a()');  
  }, 1000);  
});  
  
promiseA  
  .then((result) => {  
    console.log('promiseA success:', result);  
  }, (error) => {  
    console.log('promiseA error:', error);  
  })  
  .finally(() => {  
    console.log('a() is done!');  
  });
```

```
PS D:\Материалы\сwp_03> node .\03-09.js  
promiseA error: something bad happened a()  
a() is done!  
PS D:\Материалы\сwp_03>
```

Обратные вызовы промисов помещаются в очередь микрозадач (microtask queue)

```
setTimeout(() => console.log('setTimeout callback'), 0);

// immediately resolved promise
const promiseA = new Promise((resolve) => resolve());

// sync
console.log('I am sync job.');
```



```
// promise listener
promiseA.then(() => {
|   console.log('promiseA success:');
| });
```



```
// sync
console.log('I am good sync job.');
```

```
console.log('I am awesome sync job too.');
```

```
PS D:\Материалы\свр_03> node .\03-10.js
I am sync job.
I am good sync job.
I am awesome sync job too.
promiseA success:
setTimeout callback
PS D:\Материалы\свр_03> □
```


Цепочка промисов

Обработчик промиса возвращает **новый промис** с неопределенным result (можно его задать с помощью оператора return), поэтому их можно объединять в цепочки.

```
Promise.reject('Reject DATA!')
  .then((result) => {
    console.log('[1] then', result); // won't be called
    return '[2] then payload';
  })
  .finally(() => {
    console.log('[1] finally'); // first finally will be called
    return '[1] finally payload';
  })
  .then((result) => {
    console.log('[2] then', result); // won't be called
    return '[2] then payload';
  })
  .catch((error) => {
    console.log('[1] catch', error); // first catch will be called
    return '[1] catch payload';
  })
  .catch((error) => {
    console.log('[2] catch', error); // won't be called
    return '[2] catch payload';
  })
  .then((result) => {
    console.log('[3] then', result); // will be called
    return '[3] then payload';
  })
  .finally(() => {
    console.log('[2] finally'); // will be called
    return '[2] finally payload';
  })
  .catch((error) => {
    console.log('[3] catch', error); // won't be called
    return '[3] catch payload';
  })
  .then((result) => {
    console.log('[4] then', result); // will be called
    return '[4] then payload';
  });
```

```
PS D:\Материалы\свр_03> node .\03-11.js
[1] finally
[1] catch Reject DATA!
[3] then [1] catch payload
[2] finally
[4] then [3] then payload
PS D:\Материалы\свр_03> █
```

```

const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

Promise.all([a(), b(), c(), { key: 'I am plain data!' }])
  .then((data) => {
    console.log('success: ', data);
  })
  .catch((error) => {
    console.log('error: ', error);
  });

```

Метод **Promise.all (iterable)** возвращает промис, который выполнится **после выполнения всех промисов** в передаваемом итерируемом аргументе.

```

PS D:\Материалы\свр_03> node 03-12.js
success: [
  'result of a()',
  'result of b()',
  'result of c()',
  { key: 'I am plain data!' }
]
PS D:\Материалы\свр_03>

```

```
const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

Promise.race([a(), b(), c()])
  .then((data) => {
    console.log('success: ', data);
  })
  .catch((error) => {
    console.log('error: ', error);
  });
```

Метод **Promise.race(iterable)** возвращает промис, который будет выполнен или отклонен с **результатом** исполнения **первого выполненного или отклонённого** итерируемого промиса.

```
PS D:\Материалы\свр_03> node 03-12.js
success: result of b()
PS D:\Материалы\свр_03> █
```

async/await

ES8 / ES2017

=

синтаксис для обработки
нескольких промисов **в режиме
синхронного кода.**

Async/await

async – перед объявлением функции, возвращает промис

await – блокирует код до тех пор, пока промис не будет разрешен или отклонен.

```
async function myFunction() {  
  var result = await new MyPromise();  
  console.log(result);  
}  
myFunction(); // returns a promise
```


Async/await

```
const a = () => new Promise(resolve => {
  setTimeout(() => resolve('result of a()'), 1000);
});

const b = () => new Promise((resolve, reject) => {
  setTimeout(() => resolve('result of b()'), 500);
  //setTimeout(() => reject('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

const doJobs = async () => {
  var resultA = await a();
  var resultB = await b();
  var resultC = await c();

  return [resultA, resultB, resultC];
};

// doJobs() returns a promise
doJobs().then((result) => {
  console.log('success:', result);
})
.catch((error) => {
  console.log('error:', error);
});

console.log('I am a sync operation!');
```

```
PS D:\Материалы\свр_03> node .\03-13.js
I am a sync operation!
success: [ 'result of a()', 'result of b()', 'result of c()' ]
PS D:\Материалы\свр_03>
```

```
PS D:\Материалы\свр_03> node .\03-13.js
I am a sync operation!
error: result of b()
PS D:\Материалы\свр_03>
```

Async/await (обработка Error)

```
const a = () => new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('An error occurred')), 1000);
});

const b = () => new Promise(resolve => {
  setTimeout(() => resolve('result of b()'), 500);
});

const c = () => new Promise(resolve => {
  setTimeout(() => resolve('result of c()'), 1100);
});

const doJobs = async () => {
  try {
    var resultA = await a();
    var resultB = await b();
    var resultC = await c();

    console.log('Success: ', [resultA, resultB, resultC]);
  } catch (error) {
    console.log('error: ', error.message)
  }
};

doJobs();

console.log('I am a sync operation!');
```

Блок catch обработает только reject(new Error(...)). На reject("") блок catch не работает

```
const doJobs = async () => {
  var resultA = await a();
  var resultB = await b();
  var resultC = await c();

  return [resultA, resultB, resultC];
};

doJobs().then((result) => {
  console.log('success:', result);
})
.catch((error) => {
  console.log('error:', error.message);
});

console.log('I am a sync operation!');
```

```
PS D:\NodeJS\samples\cw03> node 03-23
I am a sync operation!
error: An error occurred
```