JavaScript
(часть 3)

Татаринова А.Г., каф. ПМИ

2025

Объявление функции

- Функции являются основными «строительными блоками» программы

```
function имя(параметры) {
  ...тело...
}
```

- Пример:

```
function showMessage() {
  alert( 'Всем привет!' );
}
showMessage();
showMessage();
```

Локальные переменные

- Переменные, объявленные внутри функции, видны только внутри этой функции

```
function showMessage() {
  let message = "Привет, я JavaScript!";
  alert( message );
}
showMessage(); // Привет, я JavaScript!
alert( message ); // <-- будет ошибка
```

Внешние переменные

- Функция обладает полным доступом к глобальным переменным и может изменять их значение
- Глобальная переменная используется, только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю
- Желательно сводить использование глобальных переменных к минимуму. В современном коде обычно мало или совсем нет глобальных переменных. Хотя они иногда полезны для хранения важнейших «общепроектовых» данных

```
let userName = 'Вася';
function showMessage() {
  let userName = "Петя";
  let message = 'Привет, ' + userName; // Петя
  alert(message);
}
showMessage();
alert( userName ); // значение «Вася» не изменилось
```

Параметры (аргументы) функции

- Можно передать внутрь функции любые данные, используя параметры (*аргументы функции*)
- Когда функция вызывается, то переданные значения копируются в соответствующие локальные переменные и используются в теле функции

```
function showMessage(from, text) {
  from = '*' + from + '*'; // немного украсим "from"
  alert( from + ': ' + text );
}
let from = "Аня";
showMessage(from, "Привет"); // *Аня*: Привет
// значение "from" осталось прежним
alert( from ); // Аня
```

Параметры по умолчанию

- Если параметр не указан, то его значением становится **`undefined`**
- Можно указать значение по умолчанию посредством «**=**»

```
function showMessage(from, text = "текст не добавлен") {
  alert( from + ": " + text );
}
showMessage("Аня"); // Аня: текст не добавлен
```

- На месте строки «*текст не добавлен*» может быть выражение, которое бы вычислялось и присваивалось при отсутствии параметра
- В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра

Параметры по умолчанию в ранних версиях

- Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах

1.
```
function showMessage(from, text) {
  if (text === undefined) { text = 'текст не добавлен';  }
  alert( from + ": " + text );
  ...
}
```

2.
```
function showMessage(from, text) {
  text = text || 'текст не добавлен';
  ...
}
```

Возврат значения

- Директива **`return`** может находиться в любом месте тела функции
- Когда выполнение тела функции доходит до **`return`**, то выполнение функции останавливается, и значение возвращается в вызвавший её код
- Вызовов **`return`** может быть несколько
- Возможно использовать **`return`** и без значения – приведёт к немедленному выходу из функции
- Пустой **`return`** аналогичен **`return undefined`**
- (!) Не добавляйте «переход на новую строку» между **`return`** и его значением

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```

- Можно

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

Function Declaration и Function Expression

- Синтаксис «объявление функции» (function declaration)

```
function sayHi() {
  alert( "Привет" );
}
```

- Синтаксис «функциональное выражение» (function expression)

```
let sayHi = function() {
  alert( "Привет" );
};
```

Function Expression создаётся здесь как `function(...) {...}` внутри выражения присваивания: `let sayHi = …;`.
Точку с запятой `;` рекомендуется ставить в конце выражения, она не является частью синтаксиса функции

- Не имеет значение как определить функцию, т.к. это просто значение, хранимое в переменной

```
alert( sayHi ); // выведет код функции
```

Function Declaration и Function Expression

- В JavaScript функции – это значение особого типа
- С функциями можно делать то же самое, что и с любым другим значением

```
function sayHi() {   // (1) создаём
  alert( "Привет" );
}
let func = sayHi;    // (2) копируем
func();              // (3) вызываем копию
sayHi();             // прежняя тоже работает
alert(sayHi)         // вывод тела функции
```

Function Declaration vs Function Expression

- Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться
- Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке)

```
sayHi("Вася"); // Привет, Вася
function sayHi(name) {
  alert( `Привет, ${name}` );
}

sayHi("Вася"); // ошибка!
let sayHi = function(name) {  // (*) магии больше нет
  alert( `Привет, ${name}` );
};
```

Стрелочные функции

- Анонимные функции с особым синтаксисом, которые принимают фиксированное число аргументов и работают в контексте включающей их области видимости, то есть — в контексте функции или другого кода, в котором они объявлены

```
(argument1, argument2, ... argumentN) => {
  // тело функции
}
```

- Если тело функции представлено единственным выражением, то это позволяет обойтись без фигурных скобок, обрамляющих тело функции
- Это сокращенная запись для

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

Примеры стрелочных функций

- Сумма двух аргументов:
```
add = (a, b) => a + b;
```
- Возврат первого элемента массива
```
getFirst = (array) => array[0];
```
- Возврат из функции объектного литерала
```
(name, description) => ({name: name, description: description});
```
- Приведение к нижнему регистру строк в массиве
```
words = ['hello', 'WORLD', 'Whatever'];
downcasedWords = words.map((word) => word.toLowerCase());
```
- Получение значения свойства объектов из массива
```
names = objects.map((object) => object.name);
```

Многострочные стрелочные функции

- Если нужна более сложная функция, с несколькими выражениями и инструкциями. Это также возможно, нужно лишь заключить их в фигурные скобки
- Если используются фигурные скобки, то важно слово **return**
- Пример:

```
let sum = (a, b) => {
  let result = a + b;
  return result;
};
alert( sum(1, 2) ); // 3
```

Колбэки

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
function showOk() {
  alert( "Вы согласны." );
}
function showCancel() {
  alert( "Вы отменили выполнение." );
}
ask("Вы согласны?", showOk, showCancel);
```

Передаём функцию и ожидаем, что она вызовется обратно (от англ. «call back» – обратный вызов) когда-нибудь позже, если это будет необходимо

В нашем случае, **`showOk`** становится колбэком для ответа «yes», а **`showCancel`** – для ответа «no»

Колбэки

```
function ask(question, yes, no) {
  if (confirm(question)) yes()
  else no();
}
ask(
  "Вы согласны?",
  function() { alert("Вы согласились."); },
  function() { alert("Вы отменили выполнение."); }
);
```

- Здесь функции объявляются прямо внутри вызова `ask(...)`. У них нет имён, поэтому они называются анонимными

Методы объекта

- Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее

```
let user = { name: "Джон",   age: 30 };
```

- Действия объекта представлены свойствами-функциями объекта

```
let user = { name: "Джон",   age: 30 };
user.sayHi = function() {
  alert("Привет!");
};
user.sayHi(); // Привет!
```

- Функцию, которая является свойством объекта, называют *методом* этого объекта
- Можно заранее объявить функцию и использовать её в качестве метода

Сокращенная запись метода

- Пример:

```
user = {
  sayHi: function() {
    alert("Привет");
  }
};
```

- Сокращённая запись

```
user = {
  sayHi() { // то же самое, что и "sayHi: function()"
    alert("Привет");
  }
};
```

Функция-конструктор

Функции-конструкторы технически являются обычными функциями
Но есть два соглашения:

- Имя функции-конструктора должно начинаться с большой буквы
- Функция-конструктор должна выполняться только с помощью оператора "**new**"

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
```

Функция-конструктор

Когда функция вызывается как **`new User(...)`**, происходит следующее:

- Создаётся новый пустой объект, и он присваивается **`this`**
- Выполняется тело функции. Обычно оно модифицирует **`this`**, добавляя туда новые свойства
- Возвращается значение **`this`**

Другими словами, **`new User(...)`** делает что-то вроде:

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}
let user = new User("Jack");
```

```
function User(name) {
  // this = {};  (неявно)

  // добавляет свойства к this
  this.name = name;
  this.isAdmin = false;

  // return this;  (неявно)
}
```

Синтаксис «new Function»

- Ещё один вариант объявлять функции

```
let func = new Function([arg1, arg2, ...argN], functionBody);
```

- Функция создаётся полностью «на лету» из строки, переданной во время выполнения. Например, можно получить новую функцию с сервера и затем выполнить её
- Это используется в очень специфических случаях, например, когда получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях
- Такая функция имеет доступ только к глобальным переменным

this

- Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода)
- Для доступа к информации внутри объекта метод может использовать ключевое слово **this**

```
let user = {  name: "Джон",  age: 30,
  sayHi() {
    alert(this.name);  // this - это "текущий объект"
  }
};
user.sayHi(); // Джон
```

- Значение **this** – это объект «перед точкой», который используется для вызова метода
- Значение **this** — это ссылка на определенный контекст внутри объекта

this

- Технически также возможно получить доступ к объекту без ключевого слова **this**, ссылаясь на него через внешнюю переменную, в которой хранится ссылка на этот объект
- Но такой код будет ненадёжным

```
let user = {  name: "Джон",  age: 30,
  sayHi() {
    alert( user.name ); // приведёт к ошибке
  }};
let admin = user;
user = null; // обнулим переменную для наглядности, теперь она не хранит ссылку на объект.
admin.sayHi(); // Ошибка! Внутри sayHi() используется user, которая больше не ссылается на объект!
```

this

- В JavaScript ключевое слово «**this**» ведёт себя иначе, чем в большинстве других ЯП. Оно может использоваться в любой функции
- Значение **this** вычисляется во время выполнения кода и зависит от контекста

```
let user = { name: "Джон" };
let admin = { name: "Админ" };
function sayHi() {  alert( this.name );}
user.f = sayHi;
admin.f = sayHi;
// вызовы функции, приведённые ниже, имеют разное значение this
user.f(); // Джон  (this == user)
admin['f'](); // Админ (неважен способ доступа к методу - через точку или квадратные скобки)
```

this

- В JavaScript ключевое слово «**this**» ведёт себя иначе, чем в большинстве других ЯП. Оно может использоваться в любой функции
- Значение **this** вычисляется во время выполнения кода и зависит от контекста

```
let user = { name: "Джон" }; let admin = { name: "Админ" };

function sayHi() {  alert( this.name );}

user.f = sayHi;

admin.f = sayHi;

// вызовы функции, приведённые ниже, имеют разное значение this

user.f(); // Джон  (this == user)

admin['f'](); // Админ (неважен способ доступа к методу - через точку или квадратные скобки)
```

при вызове `obj.f()` значение `this` внутри `f` равно `obj`

this и стрелочные функции

- У стрелочных функций нет `this`. Если происходит обращение к `this`, то его значение берётся снаружи
- Отсутствие `this` приводит к ограничению: стрелочные функции не могут быть использованы как конструкторы
- Пример:

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};
user.sayHi(); // Ilya
```

this и стрелочные функции

- Пример итерации внутри метода

```
let group = {
  title: "Our Group",
  students: ["John", "Pete", "Alice"],
  showList() {
    this.students.forEach(
      student => alert(this.title + ': ' + student)
    );
  }
};
group.showList();
```

this и стрелочные функции

- Ещё пример итерации внутри метода

```
function Group(name, students){
            this.title = name;
            this.students = students,
            this.showList =function(){ this.students.forEach(
                student => console.log(this.title + ': ' + student)
              )};
}
g1 = new Group("a", [1, 2, 3, 4]);
g1.showList();
g1.students.push(5);
g1.showList();

g2 = new Group("b", ["A", "B", "C"]);
g2.showList();
```

Декораторы

- Декоратор – это обёртка вокруг функции, которая изменяет поведение этой функции

```
function slow(x) {
  alert(`Called with ${x}`); // здесь могут быть ресурсоёмкие вычисления
  return x;}
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {    // если кеш содержит такой x,
      return cache.get(x); // читаем из него результат
    }
    let result = func(x); // иначе, вызываем функцию
    cache.set(x, result); // и кешируем (запоминаем) результат
    return result;
  };}
slow = cachingDecorator(slow);
alert( slow(1) ); // slow(1) кешируем
alert( "Again: " + slow(1) ); // возвращаем из кеша
alert( slow(2) ); // slow(2) кешируем
alert( "Again: " + slow(2) ); // возвращаем из кеша
```

Декораторы

- При создании декоратора для метода объекта есть нюансы – нужно передать контекст
- `func.call(context, arg1, arg2…)` – вызывает `func` с данным контекстом и аргументами
- `func.apply(context, args)` – вызывает `func`, передавая `context` как `this` и псевдомассив `args` как список аргументов

Пример:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}
let user = { name: "John" };
// 'user' становится 'this', и "Hello" становится первым аргументом
say.call( user, "Hello" ); // John: Hello
```

Декораторы

```
let worker = {
  someMethod() {    return 1;  },
  slow(x) {
    alert("Called with " + x); // здесь тяжёлая задача для процессора
    return x * this.someMethod(); // (*)
  }};
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };}
alert( worker.slow(1) ); // оригинальный метод работает
worker.slow = cachingDecorator(worker.slow); // теперь сделаем его кеширующим
alert( worker.slow(2) ); // Ой! Ошибка: не удаётся прочитать свойство 'someMethod'
из 'undefined'
```

Декораторы

```
let worker = {
  someMethod() {    return 1;  },
  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }};
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);    }
    let result = func.call(this, x); // теперь 'this' передаётся правильно
    cache.set(x, result);
    return result;
  };}
worker.slow = cachingDecorator(worker.slow); // теперь сделаем её кеширующей
alert( worker.slow(2) ); // работает
alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кешируется)
```

Декораторы

```
let worker = {
  slow(min, max) {
    return min + max; // здесь может быть тяжёлая задача
  }};
worker.slow = cachingDecorator(worker.slow);
```

Как использовать оба аргумента `min` и `max` для ключа в коллекции `cache`. Можно:

- Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный Map, и поддерживает множественные ключи.
- Использовать вложенные коллекции: cache.set(min) будет Map, которая хранит пару (max, result). Тогда получить result мы сможем, вызвав cache.get(min).get(max).
- **Соединить два значения в одно. В нашем случае можно просто использовать строку "min,max" как ключ к Map. Для гибкости, можно позволить передавать хеширующую функцию в декоратор, которая знает, как сделать одно значение из многих**

Декораторы

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;  }};
function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);     }
    let result = func.call(this, ...arguments); // (**)
    cache.set(key, result);
    return result;
  };}
function hash(args) {
  return args[0] + ',' + args[1];}
worker.slow = cachingDecorator(worker.slow, hash);
alert( worker.slow(3, 5) ); // работает
alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)
```

func.apply(this, arguments)

Декораторы

- Различие в синтаксисе между **`call`** и **`apply`** состоит в том, что **`call`** ожидает список аргументов, в то время как **`apply`** принимает псевдомассив.
- Эти два вызова почти эквивалентны:
  - **`func.call(context, ...args);`** // передаёт массив как список с оператором расширения
  - **`func.apply(context, args);`** // тот же эффект
- Смысловая разница:
  - Оператор расширения **`...`** позволяет передавать перебираемый объект **`args`** в виде списка в **`call`**.
  - А **`apply`** принимает только псевдомассив **`args`**.
- Оба вызова дополняют друг друга. Для перебираемых объектов сработает **`call`**, а где мы ожидаем псевдомассив – **`apply`**.
- Если у нас объект, который и то, и другое, например, реальный массив, то технически можно использовать любой метод, но **`apply`**, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше.

Асинхронность

- Многие действия в JavaScript асинхронные

Пример:

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

- Данная функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция `<script src="…">`, браузер загрузит скрипт и выполнит его
- 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

Асинхронность

- Как использовать новый скрипт, как только он будет загружен?
- Можно передать 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);
}
```

- Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передаётся аргумент `callback` — функция, которая будет вызвана по завершению асинхронного действия

Асинхронность

- Перехват ошибок
  Пример:

```
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);
}
```

- Вызов

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

Асинхронность

```
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 (*)
          }
        });
      }
    })
  }});
```

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

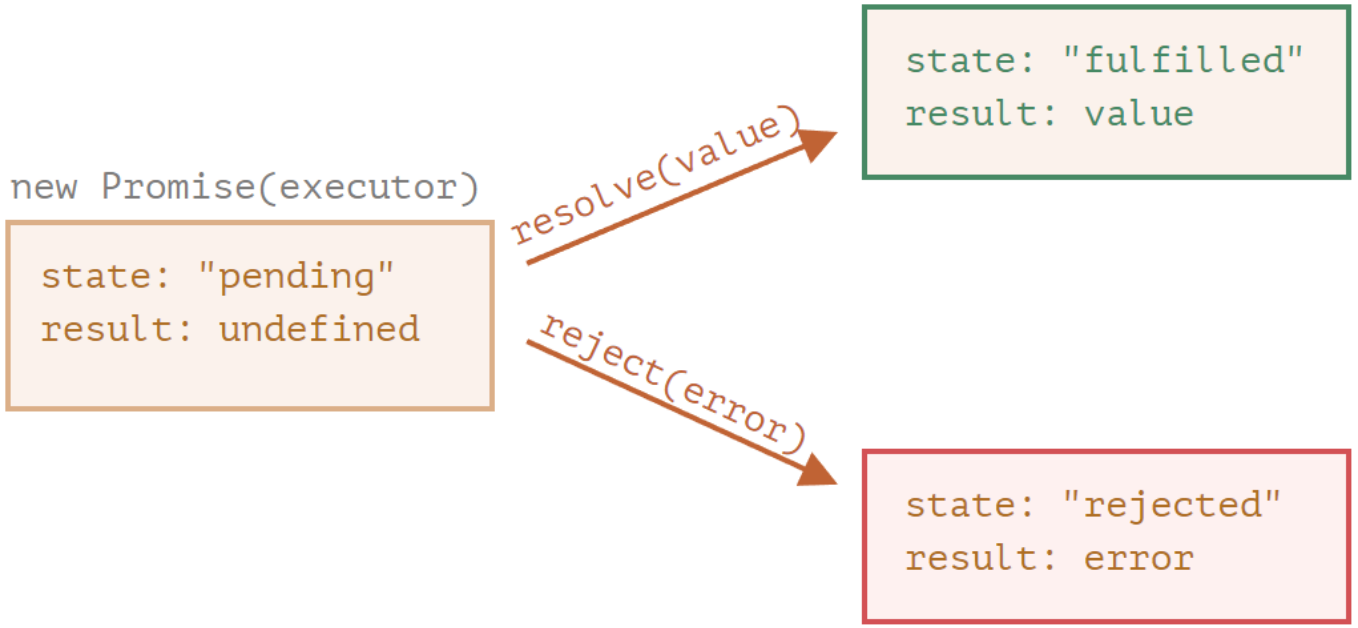
function step1(error, script) {
  if (error) {    handleError(error);
  } else {    // ...
    loadScript('2.js', step2);
  }}
function step2(error, script) {
  if (error) {    handleError(error);
  } else {    // ...
    loadScript('3.js', step3);
  }}
function step3(error, script) {
  if (error) {    handleError(error);
  } else {
    // ...и так далее, пока все скрипты не
будут загружены (*)
  }};
```

Промисы

- Promise – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе
- ```
  let promise = new Promise(function(resolve, reject) {
      ...
  });
  ```
- 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
- Её аргументы **`resolve`** и **`reject`** – это колбэки, 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
  ```

Промисы

- Исполнитель запускается автоматически, он должен выполнить работу, а затем вызвать **`resolve`** или **`reject`**
- У объекта promise, возвращаемого конструктором **`new Promise`**, есть внутренние свойства:
  - state («состояние») — вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject
  - result («результат») — вначале **`undefined`**, далее изменяется на **`value`** при вызове resolve(value) или на error при вызове reject(error)



Промисы

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

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

- 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

Промисы

- Пример с resolve

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});
promise.then(
  result => alert(result), // выведет "done!" через одну секунду
  error => alert(error) // не будет запущена);
```

- Пример с reject

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
promise.then(
  result => alert(result), // не будет запущена
  error => alert(error) // выведет "Error: Whoops!" спустя одну секунду);
```

Промисы

- Если нужно только обработать ошибку, то можно использовать `null` в качестве первого аргумента: `.then(null, errorHandlingFunction)`
- Или можно воспользоваться методом `.catch(errorHandlingFunction)`, который сделает то же самое

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});
// .catch(f) это то же самое, что promise.then(null, f)
promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

async и await

- **async** – функция, которая возвращает промис
  Пример:

```
async function f() {
  return 1;
}
f().then(alert); // 1
```

- Ключевое слово **async** перед функцией гарантирует, что эта функция в любом случае вернёт промис
- Ключевое слово **await** заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от **await** не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится
- **await** заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п.
- Это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем **promise.then**
- **await** работает только внутри **async**–функций

async и await

```
async function f() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });
  let result = await promise; // будет ждать, пока промис не выполнится (*)
  alert(result); // "готово!"
}
f();
```

fetch

- JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости
  Например:
  - Например, мы можем использовать сетевой запрос, чтобы:
  - Отправить заказ,
  - Загрузить информацию о пользователе,
  - Запросить последние обновления с сервера
- Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (Asynchronous JavaScript And XML)
  (!) можно не использовать XML
- Метод `fetch()` позволяет делать сетевые запросы и получать информацию с сервера

fetch

- Основной синтаксис
  `let promise = fetch(url, [options])`
- где
  - `url` – URL для отправки запроса
  - `options` – дополнительные параметры: метод, заголовки и т.д.
- Без `options` это простой GET-запрос, скачивающий содержимое по адресу url
- Браузер начинает запрос и возвращает промис, который внешний код использует для получения результата
- Процесс получения ответа обычно происходит в два этапа:
  - Во-первых, `promise` выполняется с объектом встроенного класса `Response` в качестве результата, как только сервер пришлёт заголовки ответа
  - Во-вторых, для получения тела ответа нам нужно использовать дополнительный вызов метода Можно выбрать только один метод чтения ответа

fetch

```
let response = await fetch(url);
if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

fetch

```
<!DOCTYPE HTML><html><body><script>
async function get_commits_author(i){
	//https://github.com/javascript-tutorial/ru.javascript.info/commits
	let url = 'https://api.github.com/repos/javascript-tutorial/ru.javascript.info/commits';
	let response = await fetch(url);
	let commits = await response.json();
	console.log(commits[i].author.login);
}
get_commits_author(0);
get_commits_author(29);
</script></body></html>
```