# Front-end

JavaScript

# JS.

- Каррирование функций
- Частичное применение функций
- Функции высшего порядка
- Колбэк функции (функции обратного вызова)
- IIFE

# JS.

## Каррирование

Каррированная функция — это функция, которая продолжает возвращать функции до тех пор, пока не будут отправлены все ее параметры.

Каррирование всегда приводит к замыканию. Потому что каждая функция, возвращаемая каррированной функцией, будет снабжена внутренним состоянием родителей.

# JS.

## Для чего нужно каррировать функции?

Каррирование делает код:

- Чище
- Уменьшает количество повторяющихся параметров и делает код более лаконичным
- Более компонуемым
- Переиспользуемым

# JS.

Каррирование или карринг (**currying**) в функциональном программирование — это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом.

При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента.

Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции — до тех пор, пока функция не получит все необходимые ей аргументы.

Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Говоря о каррировании, можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

**Арность** — это количество аргументов функции.

# JS.

У нас имеется следующая функция:

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

Она принимает три аргумента и возвращает их произведение:

```
multiply(1,2,3); // 6
```

Теперь подумаем о том, как преобразовать её к набору функций, каждая из которых принимает один аргумент.

# JS.

Создадим каррированный вариант этой функции и посмотрим на то, как получить тот же результат в ходе вызова нескольких функций:

```
function multiply(a) {
   return (b) => {
      return (c) => {
         return a * b * c
      }
   }
}
log(multiply(1)(2)(3)) // 6
```

# JS.

**Currying** - это процесс получения функции, которая обычно получает более одного аргумента (такая как add) и рефакторинга её так, что она становится функцией более высокого порядка. Карри-функция возвращает серию функций, каждая из которых принимает только один аргумент, и выполняет подсчет только после получения последнего аргумента.

```
// До
function add(x, y) {
  return x + y;
}

// После
function add(x) {
  return function(y) {
    return x + y;
  };
}
```

# JS.

При каррировании число вложенных функций равно числу аргументов исходной функции. Каждая из этих функций ожидает собственный аргумент.

Если функция аргументов не принимает, или принимает лишь один аргумент, то каррировать её нельзя.

# JS.

## Почему каррирование делает код лучше?

Некоторые функции принимают конфигурацию в качестве входных данных

Если у нас есть функции, которые принимают конфигурацию (начальные настройки), нам лучше их каррировать, потому что эти конфигурации, вероятно, будут повторяться снова и снова.

Предположим, что у нас есть функция перевода, которая принимает локаль и текст для перевода.

```
const translate = (locale, text) => { /* логика перевода */ }
```

Использование будет выглядеть так.

```
translate('ru', 'Hello')

translate('ru', 'Goodbye')

translate('ru', 'How are you?')
```

# JS.

Каждый раз, когда мы вызываем translate, мы должны указывать язык и текст. Что является избыточным для предоставления локали при каждом вызове.

Но вместо этого давайте каррируем translate следующим образом.

```
const translateCurry = (locale) => {
   return function (text) {
      console.log(`${locale}`, `${text}`)
   }
}
const translateToRu = translateCurry('ru');
const translateToDe = translateCurry('de')


translateToRu('Hello');
translateToDe('How are you?')
```

# JS.

Теперь translateToRu имеет ru в качестве locale, предоставленного каррированной функции translate, и ожидает текста. Мы можем использовать это так.

translateToRu('Hello')

translateToRu('Goodbye')

translateToRu('How are you?')

Каррирование действительно внесло улучшения, нам не нужно каждый раз указывать локаль. Вместо этого каррированная функция translateToRu содержит locale из-за каррирования.

После каррирования - в этом конкретном примере код стал:

чище

менее многословным и менее избыточным.

# JS.

## Частичное применение или partial application

Каррирование и частичное применение функций может оказаться полезным в различных ситуациях. Например — при разработке небольших модулей, подходящих для повторного использования.

Частичное применение функций позволяет облегчить использование универсальных модулей. Например, у нас есть интернет-магазин, в коде которого имеется функция, которая используется для вычисления суммы к оплате с учётом скидки.

```
function discount(price, discount) {
    return price * discount
}
```

# JS.

Есть определённая категория клиентов, назовём их «любимыми клиентами», которой мы даём скидку в 10%. Например, если такой клиент покупает что-то на $500, мы даём ему скидку размером $50:

```
const price = discount(500,0.10); // $50

// $500 - $50 = $450
```

Несложно заметить, что нам, при таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:

```
const price = discount(1500,0.10); // $150

// $1,500 - $150 = $1,350

const price = discount(2000,0.10); // $200

// $2,000 - $200 = $1,800
```

# JS.

Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки. Функция discount() в нашем примере имеет два аргумента. Вот как выглядит то, во что мы её преобразуем:

```
function discount(discount) {
   return (price) => {
      return price * discount;
   }
}
const tenPercentDiscount = discount(0.1);
```

# JS.

Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть — аргумент discount, уже будет задана:

```
tenPercentDiscount(500); // $50

// $500 - $50 = $450
```

# JS.

## Функция высшего порядка (hof)

Функции высшего порядка (higher order functions) в JavaScript - необходимый строительный блок функционального программирования на любом языке.

Функция более высокого порядка выполняет по крайней мере одну (чаще обе) из следующих вещей:

- принимает функцию в качестве аргумента;
- возвращает новую функцию.

# JS.

## Функция обратного вызова (callback)

В javascript функции могут принимать другие функции в качестве аргумента, а так же возвращать их. Такие функции называются функциями высшего порядка.

В javascript любую функцию, которая передаётся внутрь как аргумент называют колбек (callback) функцией или функцией обратного вызова.

# JS.

**Функция обратного вызова (callback)**

```
function greeting(name) {
 alert('Hello ' + name);
}

function processUserInput(callback) {
 let name = prompt('Please enter your name');
 callback(name);
}

processUserInput(greeting);
```

# JS.

## Функция обратного вызова (callback)

Функцию обратного вызова (callback) в Javascript, можно представить в реальной жизни, в виде реакции автоответчика на телефонный звонок. Ведь автоответчик обязательно ответит вам в строгой последовательности - только после полученного от вас звонка. В принципе невозможна ситуация наоборот - сначала отвечает автоответчик, а потом уже поступает ему звонок.

Как бы мы примерно записали код действия "звонок - автоответчик", без всяких коллбэков? Функция call будет звонить (соединяться с сервером и ждать от него ответа), что займет какое-то время. Вторая функция answerphone будет просто играть роль автоответчика. Затем, мы по очереди вызываем обе функции, в расчёте на то, что первой отработает функция со звонком. Так бы оно и произошло, если бы не setTimeout. Мы специально создали задержку в 500 мс. для первой функции.

# JS.

**Функция обратного вызова (callback)**

```
function call(){
   setTimeout (function(){
   console.log('Я тебе звоню');
   }, 500);
}
function answerphone (){
   console.log('Оставьте ваше сообщение');
}

call();
answerphone();
```

# JS.

## Функция обратного вызова (callback)

В реальных программах, у разных функций (в зависимости от задач) может быть разное время выполнения. Этот пример как раз демонстрирует, что быстрее отработала вторая функция. А не должна была, ведь в коде, она стоит ниже.

# JS.

## Функция обратного вызова (callback)

Функция обратного вызова (callback), избавляет программу от потенциальной проблемы - отработать неправильно. Если мы хотим, чтобы в нашем коде вторая функция выполнилась бы только после первой, то следует её задать, как callback функцию. Callback функция передается в аргументах другой функции и не начнет выполнится, пока не отработает первая функция.

Перепишем наш пример с использованием callback функции. Функция callto внутри себя будет принимать какое-то имя и callback. Мы вызываем callback внутри этой функции. При вызове функции со звонком, передадим ей два параметра: имя и callback функцию.

```
function callto(name, callback){
    console.log('Привет' + ' ' + name);
    callback();
}
callto('Alex', function(){
    console.log('После сигнала..');
})
```

# JS.

**Функция обратного вызова (callback)**

```
async function pageLoader(callback) {
 const data = await fetch('/ru/docs/Glossary/Callback_function')
 callback(data)
}

function onPageLoadingFinished(pageData) {
 console.log('Page was sucessfully loaded!')
 console.log('Response:')
 console.log(pageData)
}

pageLoader(onPageLoadingFinished)
```

# JS.

## settimeout, setInterval

setTimeout() позволяет исполнить функцию через указанный промежуток времени.

setTimeout() принимает два аргумента:

функция, которая выполнится, когда таймер закончится;

время таймера в миллисекундах.

# JS.

## settimeout, setInterval

В JavaScript код выполняется по порядку сверху вниз. Если интерпретатор встречает вызов функции, то он сразу выполняет её. Но разработчику часто может понадобиться запланировать вызов функции, чтобы она выполнилась не сразу.

Запланировать одноразовое выполнение функции можно как раз с помощью setTimeout(). Это самый простой способ исполнить функцию асинхронно.

Время таймера не гарантирует, что функция будет выполнена точно в момент, когда таймер закончится.

# JS.

## settimeout, setInterval

```
setTimeout(() => {
  console.log('Я первый!')
}, 1000)

console.log('Я второй!')
```

# JS.

## settimeout, setInterval

В JavaScript код выполняется по порядку сверху вниз. Если интерпретатор встречает вызов функции, то он сразу выполняет её. Но разработчику часто может понадобиться запланировать вызов функции, чтобы она выполнилась не сразу.

Запланировать одноразовое выполнение функции можно как раз с помощью setTimeout(). Это самый простой способ исполнить функцию асинхронно.

Время таймера не гарантирует, что функция будет выполнена точно в момент, когда таймер закончится.

# JS.

## settimeout, setInterval

setInterval() позволяет регулярно исполнять функцию через указанный промежуток времени.

Раз в секунду напечатать текст в консоль:

```
const intervalId = setInterval(function() {
  console.log('Я выполняюсь каждую секунду')
}, 1000)
```

# JS.

## settimeout, setInterval

Функция setInterval() принимает два аргумента:

функцию, которая будет регулярно выполняться при истечении таймера;

время в миллисекундах между запусками функции.

# JS.

## settimeout, setInterval

В JavaScript код выполняется по порядку сверху вниз. Если интерпретатор встречает вызов функции, то он сразу выполняет её. Но разработчику может понадобится запланировать вызов функции, чтобы она выполнилась регулярно через заданные промежутки времени. Например, чтобы регулярно проверять обновления данных на сервере.

Запланировать регулярное выполнение функции по расписанию можно с помощью setInterval().

# JS.

## setTimeout, setInterval

Таймеры имеют числовой идентификатор, он хранится в браузере в списке активных таймеров.

Остановить таймаут и интервал:

clearTimeout()

clearInterval()

# JS.

## Немедленно вызываемая функция (Immediately Invoked Function Expression — IIFE)

Это конструкция, позволяющая вызывать функцию непосредственно после ее определения.

```
(function() {


 let message = "Hello";

 alert(message); // Hello


})();
```

# JS.

## Немедленно вызываемая функция (Immediately Invoked Function Expression — IIFE)

Она используется, чтобы не допустить загрязнения глобального пространства имён. Переменные, объявленные в IIFE, невидимы за пределами этой функции.