Особенности JavaScript, которые необходимо знать, чтобы освоить React

05.06.2023

В наши дни React является одной из самых популярных библиотек JavaScript. Она может использоваться для создания динамичных и отзывчивых приложений, обеспечивает лучшую производительность и легко расширяется. В основе логики лежат компоненты, которые можно повторно использовать в различных контекстах, что уменьшает необходимость писать один и тот же код несколько раз. Одним словом, с помощью React можно создавать эффективные и мощные приложения. Поэтому никогда еще не было лучшего времени, чтобы научиться создавать приложения React.

Однако без твердого понимания некоторых ключевых особенностей JavaScript создание приложений React может оказаться сложным или даже невозможным. По этой причине мы составили список функций и концепций JavaScript, которые необходимо знать перед началом работы с React. Чем лучше вы поймете эти концепции, тем легче вам будет создавать профессиональные приложения React.

Итак, вот что мы обсудим в этой статье:

JavaScript и ECMAScript

JavaScript — это популярный язык сценариев, используемый вместе с HTML и CSS для создания динамических веб-страниц. Если HTML используется для создания структуры веб-страницы, а CSS — для создания стиля и расположения ее элементов, то JavaScript — это язык, используемый для добавления поведения на страницу, то есть для создания функциональности и интерактивности. С тех пор язык был принят основными браузерами, и был написан документ, описывающий то, как должен

работать JavaScript: стандарт ECMAScript. По состоянию на 2015 год обновление стандарта ECMAScript выпускается ежегодно, и поэтому каждый год в JavaScript добавляются новые возможности.

ECMAScript 2015 стал шестым выпуском стандарта и поэтому также известен как ES6. Следующие версии обозначаются в прогрессии, поэтому мы называем ECMAScript 2016 – ES7, ECMAScript 2017 – ES8 и так далее. Из-за частоты добавления новых функций в стандарт, некоторые из них могут поддерживаться не во всех браузерах. Как же убедиться, что новейшие функции JavaScript, которые вы добавили в свое JS-приложение, будут работать так, как ожидается, во всех веб-браузерах?

У вас есть три варианта:

1. Подождать, пока все основные браузеры обеспечат поддержку новых функций. Но если вам совершенно необходима новая потрясающая функция JS для вашего приложения, этот вариант не подходит.
2. Использовать Polyfill, который представляет собой “фрагмент кода (обычно JavaScript в Web), используемый для обеспечения современной функциональности в старых браузерах, которые не поддерживают ее изначально” (см. также mdn web docs).
3. Используйте JavaScript-транспойлер, например Babel или Traceur, которые преобразуют код ECMAScript 2015+ в версию JavaScript, поддерживаемую всеми браузерами.

Утверждения и выражения

Понимание разницы между утверждениями и выражениями очень важно при создании приложений React. Итак, давайте ненадолго вернемся к основным понятиям программирования. Компьютерная программа – это список инструкций, которые должны быть выполнены компьютером. Эти инструкции называются утверждениями. В отличие от утверждений, выражения – это

фрагменты кода, которые производят значение. В операторе выражение – это часть, которая возвращает значение, и мы обычно видим его справа от знака равенства.

Оператор – это блок кода, который что-то делает.

В то время как:

Выражение – это фрагмент кода, который выдает значение.

Утверждения JavaScript могут быть блоками или строками кода, которые обычно заканчиваются точкой с запятой или заключаются в фигурные скобки.

Вот простой пример утверждения в JavaScript:

```
document.getElementById("hello").innerHTML = "Hello World!";
```

Приведенное выше утверждение записывает “Hello World!” в элемент DOM с id=”hello”. Как мы уже говорили, выражения порождают значение или сами являются значением. Рассмотрим следующий пример:

```
msg = document.getElementById("hello").value;
```

document.getElementById(“hello”).value – это выражение, поскольку оно возвращает значение.

Дополнительный пример должен прояснить разницу между выражениями и утверждениями:

```
const msg = "Hello World!";
function sayHello( msg ) {
        console.log( msg );
}
```

В приведенном выше примере,

- Первая строка является утверждением, где “Hello World!” является выражением,
- Объявление функции – это оператор, где параметр msg,

передаваемый в функцию, является выражением,

- Строка, печатающая сообщение в консоли, – это оператор, где параметр msg – это выражение.

Почему выражения важны в React

При создании приложения React вы можете вводить выражения JavaScript в код JSX. Например, вы можете передать переменную, написать обработчик события или условие. Для этого необходимо заключить JS-код в фигурные скобки.

Например, вы можете передать переменную:

```
const Message = () => {
        const name = "Carlo";
        return <p>Welcome {name}!</p>;
}
```

Короче говоря, фигурные скобки указывают вашему транспилятору обрабатывать код, заключенный в скобки, как JS-код. Все, что находится до открывающего тега и после закрывающего тега , является обычным кодом JavaScript. Все, что находится внутри открывающего и закрывающего тегов, обрабатывается как JSX-код.

Вот еще один пример:

```
const Message = () => {
        const name = "Ann";
        const heading = <h3>Welcome {name}</h3>;
        return (
                <div>
                        {heading}
                        <p>This is your dashboard.</p>
                </div>
        );
}
```

Вы также можете передать объект:

```
render(){
        const person = {
```

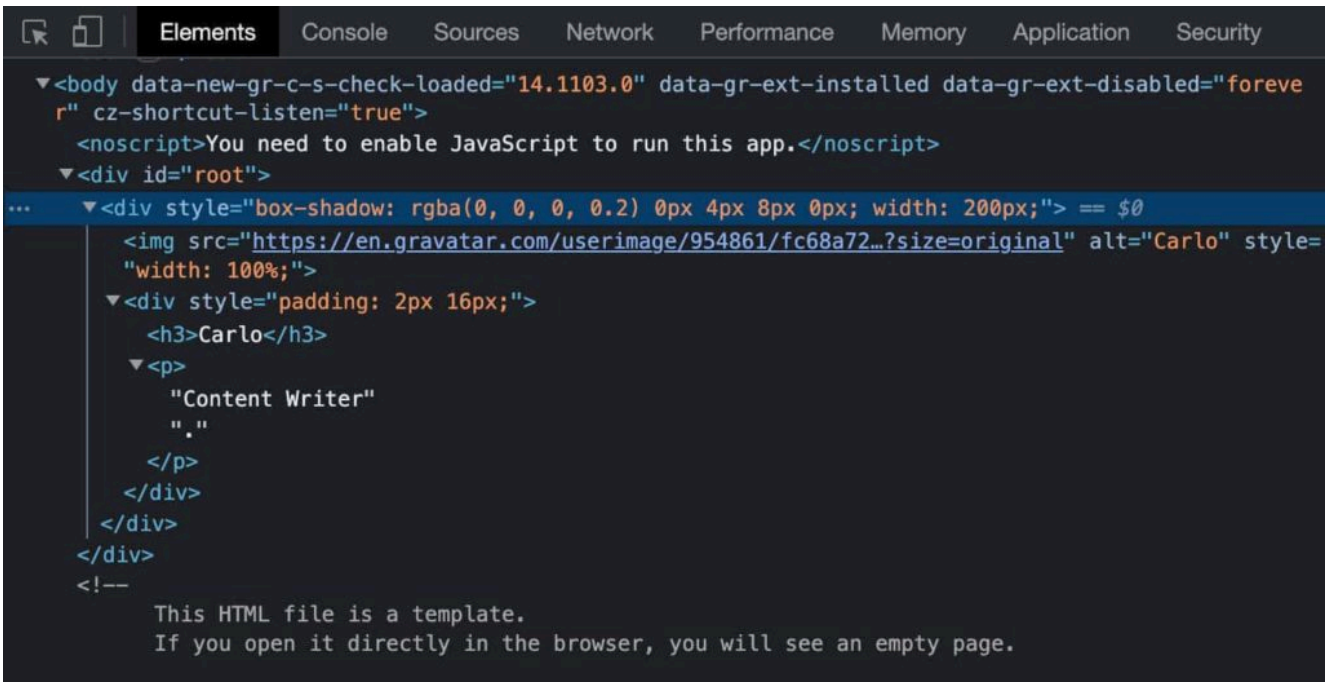
```
                name: 'Carlo',
                                                        avatar:
'https://en.gravatar.com/userimage/954861/fc68a728946aac04f853
1c3a8742ac22',
                description: 'Content Writer'
        }

        return (
                <div>
                        <h2>Welcome {person.name}</h2>
                        <img
                                className="card"
                                src={person.avatar}
                                alt={person.name}
                        />
                                                <p>Description:
{person.description}.</p>
                </div>
        );

}
```

Ниже приведен более полный пример:

```
render(){
        const person = {
                name: 'Carlo',
                                                        avatar:
'https://en.gravatar.com/userimage/954861/fc68a728946aac04f853
1c3a8742ac22?size=original',
                description: 'Content Writer',
                theme: {
                                boxShadow: '0 4px 8px 0
rgba(0,0,0,0.2)', width: '200px'
                }
        }

        return (
                <div style={person.theme}>
                        <img
                                src={person.avatar}
                                alt={person.name}
```

```
                                style={ { width: '100%' } }
                            />
                             <div style={ { padding: '2px 16px' }
}>
                                <h3>{person.name}</h3>
                                <p>{person.description}.</p>
                            </div>
                    </div>
            );
}
```

Обратите внимание на двойные фигурные скобки в атрибутах стиля в элементах img и div. Мы использовали двойные скобки для передачи двух объектов, содержащих стили карточки и изображения.



Вы должны были заметить, что во всех приведенных выше примерах мы включали выражения JavaScript в JSX. JSX принимает только выражения JavaScript в фигурных скобках. Вам не разрешается писать утверждения в коде JSX.

К ним относятся:

- Переменные
- строки с кавычками
- Вызовы функций

- Объекты
- Условные выражения

Неизменяемость в React

Мутабельность и неизменяемость — два ключевых понятия в объектно-ориентированном и функциональном программировании. Неизменяемость означает, что значение не может быть изменено после его создания. Неизменяемость означает, конечно же, обратное. В Javascript примитивные значения являются неизменяемыми, то есть после создания примитивного значения его нельзя изменить. И наоборот, массивы и объекты являются изменяемыми, поскольку их свойства и элементы могут быть изменены без переназначения нового значения.

Существует несколько причин для использования неизменяемых объектов в JavaScript:

- Повышение производительности
- Снижение потребления памяти
- Безопасность потоков
- Более простое кодирование и отладка

В соответствии с принципом неизменяемости, после того, как переменная или объект назначены, они не могут быть повторно назначены или изменены. Когда вам нужно изменить данные, вы должны создать их копию и изменить ее содержимое, оставив исходное содержимое неизменным. Неизменяемость также является ключевой концепцией в React.

В документации React говорится:

> *Состояние компонента класса доступно как this.state. Поле state должно быть объектом. Не изменяйте состояние напрямую. Если вы хотите изменить состояние, вызовите setState с новым состоянием.*

Всякий раз, когда состояние компонента изменяется, React вычисляет, нужно ли перерендерить компонент и обновить Virtual DOM. Если бы React не знал о предыдущем состоянии, он не смог бы определить, нужно ли перерисовывать компонент или нет. В документации React приведен отличный пример этого. Какие возможности JavaScript мы можем использовать, чтобы гарантировать неизменность объекта состояния в React? Давайте узнаем!

Объявление переменных

В JavaScript есть три способа объявить переменную: var, let и const. Оператор var существует с самого начала развития JavaScript. Он используется для объявления переменной, скопированной в функции или глобально скопированной, и по желанию инициализирует ее значением. Когда вы объявляете переменную с помощью var, вы можете повторно объявлять и обновлять эту переменную как в глобальной, так и в локальной области видимости. Следующий код является допустимым:

```
// Declare a variable
var msg = "Hello!";

// Redeclare the same variable
var msg = "Goodbye!"

// Update the variable
msg = "Hello again!"
```

Объявление переменных обрабатывается до выполнения кода. В результате объявление переменной в любом месте кода эквивалентно ее объявлению в самом верху. Такое поведение называется подъемом. Стоит отметить, что поднимается только объявление переменной, а не инициализация, которая происходит только тогда, когда поток управления достигает оператора присваивания. До этого момента переменная не определена:

```
console.log(msg); // undefined
var msg = "Hello!";
```

```
console.log(msg); // Hello!
```

Область видимости переменной var, объявленной в функции JS, — это все тело этой функции. Это означает, что переменная определяется не на уровне блока, а на уровне всей функции. Это приводит к ряду проблем, которые могут сделать ваш код JavaScript глючным и сложным для сопровождения. Чтобы решить эти проблемы, в ES6 было введено ключевое слово let.

> *Объявление let объявляет локальную переменную, скопированную в блок, и по желанию инициализирует ее значением.*

Каковы преимущества let перед var? Вот некоторые из них:

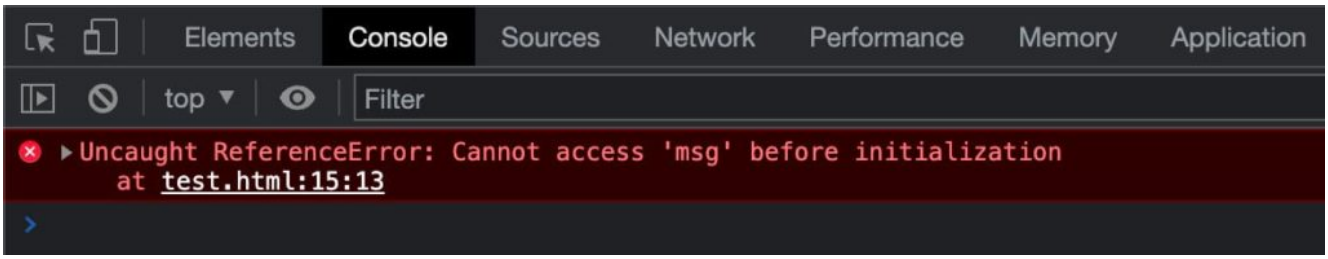
- let объявляет переменную в области видимости блока операторов, в то время как var объявляет переменную глобально или локально для всей функции независимо от области видимости блока.
- Глобальные переменные let не являются свойствами объекта window. Вы не можете получить к ним доступ с помощью window.variableName.
- К переменной let можно обратиться только после того, как будет достигнуто ее объявление. Переменная не инициализируется до тех пор, пока поток управления не достигнет строки кода, где она объявлена (объявления let не являются списочными).
- Повторное объявление переменной с помощью let приводит к ошибке синтаксиса.

Поскольку переменные, объявленные с помощью var, не могут быть блочно скопированы, если вы определите переменную с помощью var в цикле или внутри оператора if, доступ к ней может быть получен вне блока, что может привести к ошибкам в коде.

Код в первом примере выполняется без ошибок. Теперь замените var на let в блоке кода, показанном выше:

```
console.log(msg);
let msg = "Hello!";

console.log(msg);
```

Во втором примере использование let вместо var приводит к возникновению Uncaught ReferenceError:



Screenshot

Поэтому, как правило, всегда следует использовать let вместо var.

В ES6 также появилось третье ключевое слово: const.

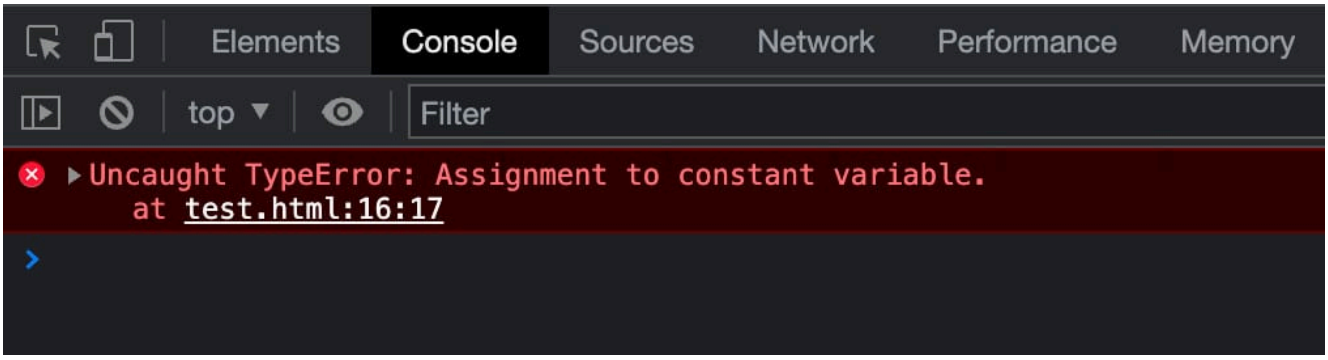
const очень похоже на let, но с ключевым отличием:

Переменным, объявленным с const, нельзя присвоить значение, кроме как в момент их объявления.

Рассмотрим следующий пример:

```
const MAX_VALUE = 1000;

MAX_VALUE = 2000;
```

Приведенный выше код выдаст следующую ошибку TypeError:

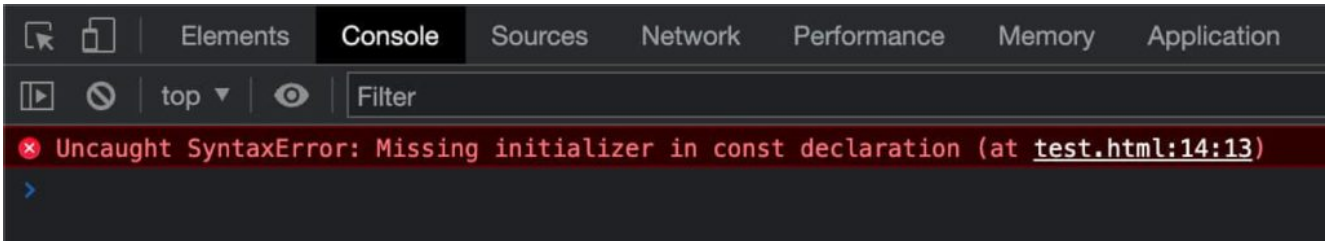


Screenshot

Кроме того:

Вы не можете объявить const, не придав ей значения.

Объявление const без указания его значения приведет к следующей синтаксической ошибке (см. также ES6 In Depth: let и const):



Screenshot

Константа не может быть объявлена заново, и ее значение не может быть изменено путем переназначения.

Но если константа является массивом или объектом, вы можете редактировать свойства или элементы внутри этого массива или объекта.

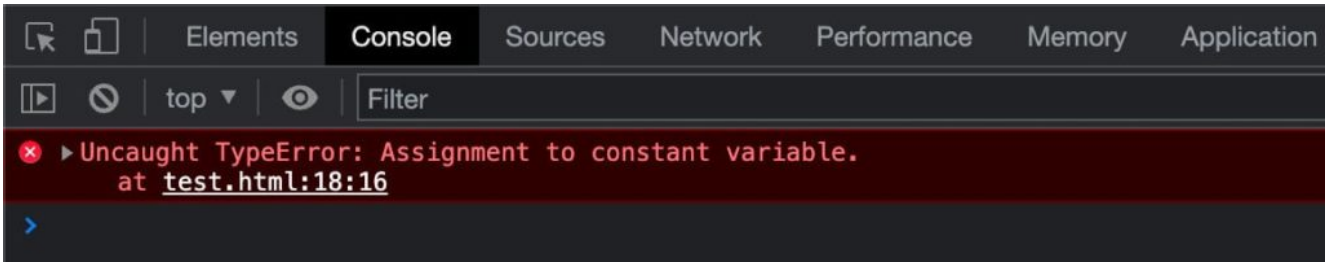
Например, можно изменять, добавлять и удалять элементы массива:

```
// Declare a constant array
const cities = ["London", "New York", "Sydney"];

// Change an item
cities[0] = "Madrid";

// Add an item
cities.push("Paris");

// Remove an item
cities.pop();

console.log(cities);

// Array(3)
// 0: "Madrid"
// 1: "New York"
// 2: "Sydney"
```

Но вам не разрешается переназначать массив:

```
const cities = ["London", "New York", "Sydney"];

cities = ["Athens", "Barcelona", "Naples"];
```

Приведенный выше код приведет к ошибке TypeError.



Screenshot

Вы можете добавлять, переназначать и удалять свойства и методы объектов:

```
// Declare a constant obj
const post = {
        id: 1,
        name: 'JavaScript is awesome',
          excerpt: 'JavaScript is an awesome scripting language',
         content: 'JavaScript is a scripting language that enables you to create dynamically updating content.'
};

// add a new property
post.slug = "javascript-is-awesome";

// Reassign property
post.id = 5;

// Delete a property
delete post.excerpt;

console.log(post);

// {id: 5, name: 'JavaScript is awesome', content: 'JavaScript is a scripting language that enables you to create dynamically updating content.', slug: 'javascript-is-awesome'}
```

Но вам не разрешается переназначать сам объект. В следующем коде произойдет Uncaught TypeError:

```
// Declare a constant obj
const post = {
        id: 1,
        name: 'JavaScript is awesome',
        excerpt: 'JavaScript is an awesome scripting language'
};

post = {
        id: 1,
        name: 'React is powerful',
        excerpt: 'React lets you build user interfaces'
};
```

В React объявление переменных с помощью const является стандартным. let следует использовать, когда const не подходит. Использование var крайне не рекомендуется.

Object.freeze()

Теперь мы согласны с тем, что использование const не всегда гарантирует сильную неизменяемость (особенно при работе с объектами и массивами). Итак, как вы можете реализовать паттерн неизменяемости в своих приложениях React? Во-первых, когда вы хотите предотвратить изменение элементов массива или свойств объекта, вы можете использовать статический метод Object.freeze().

> *Замораживание объекта предотвращает расширение и делает существующие свойства неперезаписываемыми и неконфигурируемыми. Замороженный объект больше не может быть изменен: новые свойства не могут быть добавлены, существующие свойства не могут быть удалены, их перечислимость, конфигурируемость, возможность записи или значение не могут быть изменены, а прототип объекта не может быть переназначен. freeze() возвращает тот же объект, который был передан.*

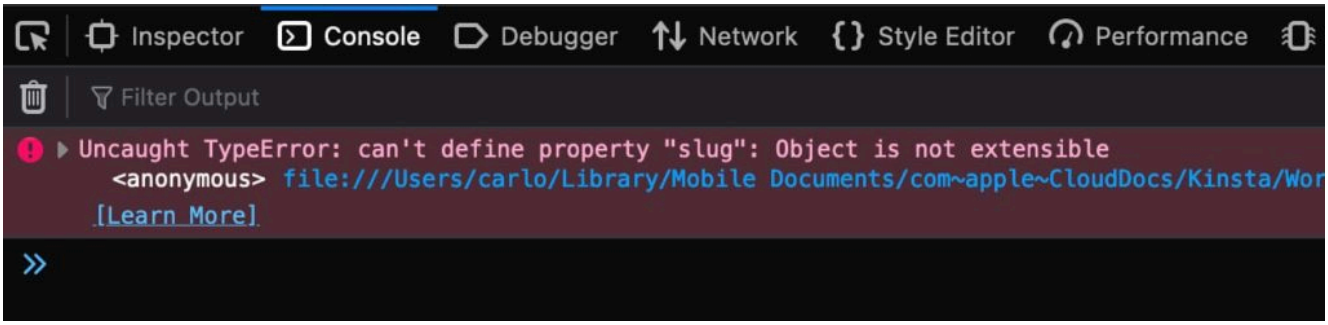
Любая попытка добавить, изменить или удалить свойство будет неудачной, либо молча, либо с выбросом TypeError, чаще всего в строгом режиме.

Вы можете использовать Object.freeze() таким образом:

```
'use strict'
// Declare a constant obj
const post = {
        id: 1,
        name: 'JavaScript is awesome',
        excerpt: 'JavaScript is an awesome scripting language'
};
// Freeze the object
Object.freeze(post);
```

Если теперь вы попытаетесь добавить свойство, вы получите ошибку Uncaught TypeError:

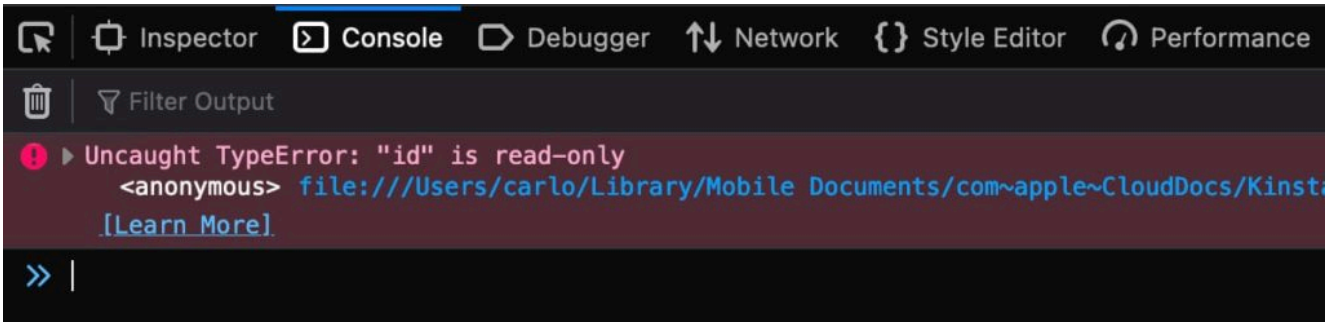
```
// Add a new property
post.slug = "javascript-is-awesome"; // Uncaught TypeError
```



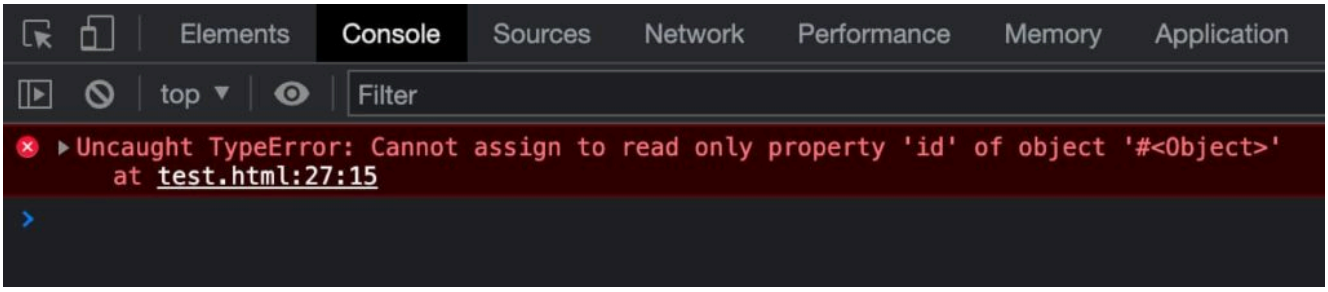
Screenshot

Когда вы пытаетесь переназначить свойство, вы получаете другой тип ошибки TypeError:

```
// Reassign property
post.id = 5; // Uncaught TypeError
```



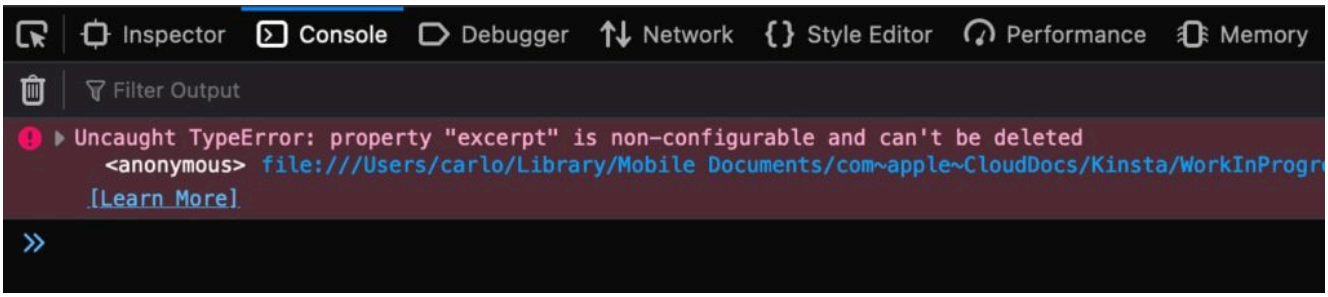
Screenshot



Screenshot

Вы также можете попытаться удалить свойство. Результатом будет еще одна ошибка TypeError:

```
// Delete a property
delete post.excerpt; // Uncaught TypeError
```



Screenshot

Шаблонные литералы

Когда вам нужно объединить строки с выводом выражений в JavaScript, вы обычно используете оператор сложения +. Однако вы также можете использовать функцию JavaScript, которая позволяет включать выражения в строки без использования оператора сложения: Шаблонные литералы. Шаблонные литералы – это особый вид строк, разделенных символами обратного тика (`). В шаблонные литералы можно включать заполнители, которые являются встроенными выражениями, разделенными символом

доллара и заключенными в фигурные скобки.

Вот пример:

```
const align = 'left';

console.log(`This string is ${ align }-aligned`);
```

Строки и заполнители передаются в функцию по умолчанию, которая выполняет интерполяцию строк для замены заполнителей и объединяет части в одну строку. Вы также можете заменить функцию по умолчанию пользовательской функцией. Шаблонные литералы можно использовать для: **Многострочных строк:** символы новой строки являются частью шаблонного литерала.

```
console.log(`Twinkle, twinkle, little bat!
How I wonder what you’re at!`);
```

Интерполяция строк: Без шаблонных литералов вы можете использовать только оператор сложения для объединения вывода выражений со строками. Смотрите следующий пример:

```
const a = 3;
const b = 7;
console.log("The result of " + a + " + " + b + " is " + (a + b));
```

Это немного запутанно, не так ли? Но вы можете написать этот код более читабельным и удобным для обслуживания способом, используя Шаблонные Литералы:

```
const a = 3;
const b = 7;
console.log(`The result of ${ a } + ${ b } is ${ a + b }`);
```

Но имейте в виду, что между этими двумя синтаксисами есть разница:

Шаблонные литералы преобразуют свои выражения непосредственно в строки, в то время как сложение преобразует свои операнды сначала в примитивы.

Шаблонные литералы имеют несколько вариантов использования. В следующем примере мы используем тернарный оператор для присвоения значения атрибуту класса.

```
const page = 'archive';

console.log(`class=${ page === 'archive' ? 'archive' : 'single' }`);
```

Ниже мы проведем простой расчет:

```
const price = 100;
const VAT = 0.22;

console.log(`Total price: ${ (price * (1 + VAT)).toFixed(2) }`);
```

Также можно вложить Шаблонные литералы, включив их внутрь заполнителя ${выражение} (но используйте вложенные шаблоны с осторожностью, так как сложные строковые структуры могут быть трудны для чтения и поддержки). **Шаблоны с метками:** Как мы уже упоминали выше, можно также определить пользовательскую функцию для выполнения конкатенации строк. Такой вид шаблонного литерала называется тегированным шаблоном.

> *Теги позволяют разбирать шаблонные литералы с помощью функции. Первый аргумент функции тега содержит массив строковых значений. Остальные аргументы относятся к выражениям.*

Теги позволяют разбирать шаблонные литералы с помощью пользовательской функции. Первый аргумент этой функции — массив строк, входящих в шаблонный литерал, остальные аргументы — выражения. Вы можете создать пользовательскую функцию, которая будет выполнять любые операции над аргументами шаблона и возвращать обработанную строку. Вот очень простой пример шаблона с тегами:

```
const name = "Carlo";
const role = "student";
```

```
const organization = "North Pole University";
const age = 25;

function customFunc(strings, ...tags) {
        console.log(strings); // ['My name is ', ", I'm ", ', and I am ', ' at ', '', raw: Array(5)]
        console.log(tags); // ['Carlo', 25, 'student', 'North Pole University']
        let string = '';
        for ( let i = 0; i < strings.length - 1; i++ ){
                console.log(i + "" + strings[i] + "" + tags[i]);
                string += strings[i] + tags[i];
        }
        return string.toUpperCase();
}

const output = customFunc`My name is ${name}, I'm ${age}, and I am ${role} at ${organization}`;
console.log(output);
```

Приведенный выше код печатает элементы массива строк и тегов, затем выводит символы строк заглавными буквами перед печатью вывода в консоль браузера.

Стрелочные функции

Стрелочные функции являются альтернативой анонимным функциям (функциям без имен) в JavaScript, но с некоторыми отличиями и ограничениями.

Все следующие объявления являются допустимыми примерами стрелочных функций:

```
// Arrow function without parameters
const myFunction = () => expression;

// Arrow function with one parameter
const myFunction = param => expression;

// Arrow function with one parameter
```

```
const myFunction = (param) => expression;

// Arrow function with more parameters
const myFunction = (param1, param2) => expression;

// Arrow function without parameters
const myFunction = () => {
        statements
}

// Arrow function with one parameter
const myFunction = param => {
        statements
}

// Arrow function with more parameters
const myFunction = (param1, param2) => {
        statements
}
```

Круглые скобки можно опустить, если в функцию передается только один параметр. Если вы передаете два или более параметров, их необходимо заключить в скобки. Вот пример:

```
const render = ( id, title, category ) => `${id}: ${title} - ${category}`;

console.log( render ( 5, 'Hello World!', "JavaScript" ) );
```

Однострочные стрелочные функции по умолчанию возвращают значение. Если вы используете многострочный синтаксис, вам придется вручную возвращать значение:

```
const render = ( id, title, category ) => {
        console.log( `Post title: ${ title }` );
        return `${ id }: ${ title } - ${ category }`;
}
console.log( `Post details: ${ render ( 5, 'Hello World!', "JavaScript" ) }` );
```

Обычно в приложениях React вы будете использовать функции Arrow Function, если нет особых причин не использовать их.

Следует помнить об одном ключевом различии между обычными функциями и функциями Arrow Functions: функции Arrow не имеют собственных привязок к ключевому слову this. Если вы попытаетесь использовать this в функции Arrow, оно выйдет за пределы области видимости функции. Для более подробного описания функций Arrow и примеров использования читайте также веб-документы mdn.

Классы

Классы в JavaScript – это особый тип функций для создания объектов, использующих механизм прототипического наследования.

Согласно веб-документам mdn,

> *Когда дело доходит до наследования, JavaScript имеет только одну конструкцию: объекты. Каждый объект имеет частное свойство, которое содержит ссылку на другой объект, называемый его прототипом. Этот объект-прототип имеет свой прототип, и так далее, пока не будет достигнут объект с null в качестве прототипа.*

Как и в случае с функциями, у вас есть два способа определения класса:

- Выражение класса
- Объявление класса

Вы можете использовать ключевое слово class для определения класса внутри выражения, как показано в следующем примере:

```
const Circle = class {
        constructor(radius) {
                this.radius = Number(radius);
        }
        area() {
                return Math.PI * Math.pow(this.radius, 2);
        }
```

```
        circumference() {
                return Math.PI * this.radius * 2;
        }
}
console.log('Circumference:       '       +       new
Circle(10).circumference()); // 62.83185307179586
console.log('Area:  '  +  new  Circle(10).area());  //
314.1592653589793
```

Класс имеет тело, то есть код, заключенный в фигурные скобки. Здесь вы определяете конструктор и методы, которые также называются членами класса. Тело класса выполняется в строгом режиме даже без использования директивы ‘strict mode’. Метод конструктора используется для создания и инициализации объекта, созданного с помощью класса, и автоматически выполняется при инстанцировании класса. Если вы не определите метод конструктора в своем классе, JavaScript автоматически использует конструктор по умолчанию. Класс может быть расширен с помощью ключевого слова extends.

```
class Book {
        constructor(title, author) {
                this.booktitle = title;
                this.authorname = author;
        }
        present() {
                return this.booktitle + ' is a great book from
' + this.authorname;
        }
}

class BookDetails extends Book {
        constructor(title, author, cat) {
                super(title, author);
                this.category = cat;
        }
        show() {
                  return this.present() + ', it is a ' +
this.category + ' book';
        }
}
```

```
const bookInfo = new BookDetails("The Fellowship of the Ring",
"J. R. R. Tolkien", "Fantasy");
console.log(bookInfo.show());
```

Конструктор может использовать ключевое слово super для вызова родительского конструктора. Если вы передадите аргумент в метод super(), этот аргумент также будет доступен в классе родительского конструктора. Для более глубокого погружения в классы JavaScript и нескольких примеров использования см. также веб-документацию mdn. Классы часто используются для создания компонентов React. Обычно вы не создаете свои собственные классы, а расширяете встроенные классы React.

Все классы в React имеют метод render(), который возвращает элемент React:

```
class Animal extends React.Component {
        render() {
                                    return <h2>Hey, I am a
{this.props.name}!</h2>;
        }
}
```

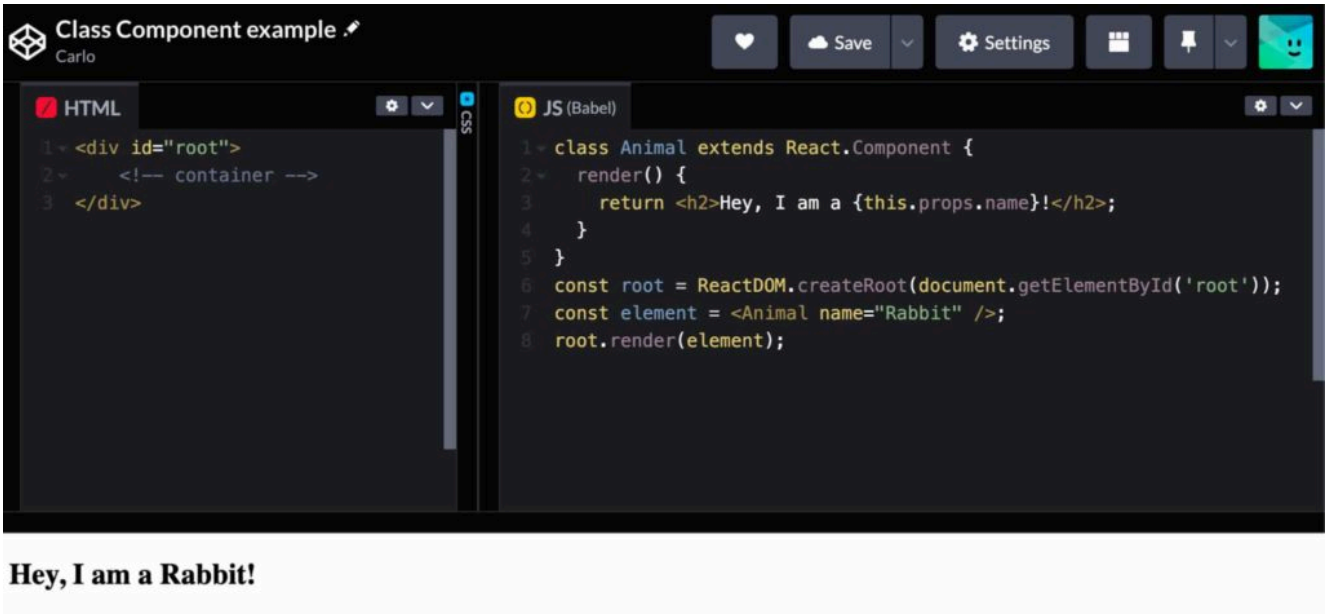
В приведенном выше примере Animal – это компонент класса. Помните, что

- Имя компонента должно начинаться с заглавной буквы
- Компонент должен включать выражение extends React.Component. Это дает доступ к методам React.Component.
- Метод render() возвращает HTML и является обязательным.

После того как вы создали свой компонент класса, вы можете вывести HTML на страницу:

```
const                              root                              =
ReactDOM.createRoot(document.getElementById('root'));
const element = <Animal name="Rabbit" />;
root.render(element);
```

На изображении ниже показан результат на странице (Вы можете увидеть его в действии на CodePen).



Screenshot

Обратите внимание, однако, что использование компонентов классов в React не рекомендуется, предпочтительнее определять компоненты как функции.

Ключевое слово ‘this’

В JavaScript ключевое слово this является общим заполнителем, обычно используемым внутри объектов, классов и функций, и относится к различным элементам в зависимости от контекста или области видимости. this может быть использовано в глобальной области видимости. Если вы напечатаете this в консоли браузера, вы получите:

Window {window: Window, self: Window, document: document, name: '', location: Location, ...}

Вы можете получить доступ к любому из методов и свойств объекта Window. Так, если вы запустите this.location в консоли браузера, вы получите следующий результат:

Location {ancestorOrigins: DOMStringList, href: 'https://kinsta.com/', origin: 'https://kinsta.com', protocol: 'https:', host: 'kinsta.com', ...}

Когда вы используете это в объекте, оно относится к самому объекту. Таким образом, вы можете ссылаться на значения объекта в методах самого объекта:

```
const post = {
        id: 5,
        getSlug: function(){
                return `post-${this.id}`;
        },
        title: 'Awesome post',
        category: 'JavaScript'
};
console.log( post.getSlug );
```

Теперь давайте попробуем использовать это в функции:

```
const useThis = function () {
        return this;
}
console.log( useThis() );
```

Если вы не соблюдаете строгий режим, вы получите:

```
Window {window: Window, self: Window, document: document, name: '', location: Location, ...}
```

Но если вы вызовете строгий режим, вы получите другой результат:

```
const doSomething = function () {
        'use strict';
        return this;
}
console.log( doSomething() );
```

В этом случае функция возвращает неопределенное значение. Это потому, что this в функции относится к ее явному значению. Как же явно задать this в функции?

Во-первых, вы можете вручную назначить функции свойства и методы:

```
function doSomething( post ) {
```

```
        this.id = post.id;
        this.title = post.title;
        console.log( `${this.id} - ${this.title}` );
}
new doSomething( { id: 5, title: 'Awesome post' } );
```

Но вы также можете использовать методы call, apply и bind, а также стрелочные функции. Метод call() функции принимает объект, на который ссылается.

```
const doSomething = function() {
        console.log( `${this.id} - ${this.title}` );
}
doSomething.call( { id: 5, title: 'Awesome post' } );
```

Метод call() может быть использован в любой функции и делает именно то, о чем говорит: вызывает функцию. Кроме того, call() принимает любой другой параметр, определенный в функции:

```
const doSomething = function( cat ) {
        console.log( `${this.id} - ${this.title} - Category:
${cat}` );
}
doSomething.call( { id: 5, title: 'Awesome post' },
'JavaScript' );
```

Метод apply() принимает объект, на который будет ссылаться функция, и массив параметров функции.

```
const doSomething = function( cat1, cat2 ) {
        console.log( `${this.id} - ${this.title} - Categories:
${cat1}, ${cat2}` );
}
doSomething.apply( { id: 5, title: 'Awesome post' },
['JavaScript', 'React'] );
```

Метод bind() связывает объект с функцией так, что при вызове функции она ссылается на объект.

```
const post = { id: 5, title: 'Awesome post', category:
'JavaScript' };
const doSomething = function() {
            return `${this.id} - ${this.title} -
```

```
${this.category}`;
}
const bindRender = doSomething.bind( post );
console.log( bindRender() );
```

Альтернативой рассмотренным выше вариантам является использование стрелочных функций.

> *Выражения стрелочных функций следует использовать только для функций, не являющихся методами, поскольку они не имеют собственного this.*

Это делает стрелочные функции особенно полезными в обработчиках событий.

Это потому, что “когда код вызывается из атрибута встроенного обработчика событий, его this устанавливается на элемент DOM, на котором размещен слушатель” (см. mdn web docs).

Но со стрелочными функциями все меняется, потому что…

> *… стрелочные функции устанавливают значение this на основе области видимости, в которой определена стрелочная функция, и значение this не меняется в зависимости от того, как вызывается функция.*

Использование стрелочных функций позволяет напрямую привязать контекст к обработчику событий.

Привязка ‘this’ к обработчикам событий в React

Когда речь идет о React, у вас есть несколько способов убедиться, что обработчик события не потеряет свой контекст:

1. Использование bind() внутри метода render:

```
import React, { Component } from 'react';
class MyComponent extends Component {
```

```
        state = { message: 'Hello World!' };

        showMessage(){
                console.log( 'This refers to: ', this );
                      console.log( 'The message is: ',
this.state.message );
        }

        render(){
                              return( <button onClick={
this.showMessage.bind( this ) }>Show message from
state!</button> );
        }
}
export default MyComponent;
```

2. Привязка контекста к обработчику события в конструкторе:

```
import React, { Component } from 'react';
class MyComponent extends Component {
        state = { message: 'Hello World!' };

        constructor(props) {
                super(props);
                this.showMessage = this.showMessage.bind( this
);
        }

        showMessage(){
                console.log( 'This refers to: ', this );
                      console.log( 'The message is: ',
this.state.message );
        }

        render(){
                  return( <button onClick={ this.showMessage
}>Show message from state!</button> );
        }
}
export default MyComponent;
```

3. Определите обработчик события с помощью стрелочных функций:

```
import React, { Component } from 'react';
class MyComponent extends Component {
        state = { message: 'Hello World!' };

        showMessage = () => {
                console.log( 'This refers to: ', this );
                        console.log( 'The message is: ',
this.state.message );
        }

        render(){
                                                return(  <button
onClick={this.showMessage}>Show message from state!</button>
);
        }
}
export default MyComponent;
```

4. Использование стрелочных функций в методе рендеринга:

```
import React, { Component } from 'react';
class MyComponent extends Component {
        state = { message: 'Hello World!' };

        showMessage() {
                console.log( 'This refers to: ', this );
                        console.log( 'The message is: ',
this.state.message );
        }

        render(){
                                                return(  <button
onClick={()=>{this.showMessage()}}>Show  message  from
state!</button> );
        }
}
export default MyComponent;
```

Какой бы метод вы ни выбрали, при нажатии на кнопку консоль браузера покажет следующий результат:

```
This refers to:  MyComponent {props: {…}, context: {…}, refs:
```

{…}, updater: {…}, state: {…}, …}

The message is: Hello World!

Тернарный оператор

Условный оператор (или тернарный оператор) позволяет писать простые условные выражения в JavaScript. Он принимает три операнда:

- Условие, за которым следует вопросительный знак (?),
- Выражение для выполнения, если условие истинно, за которым следует двоеточие (:),
- Второе выражение для выполнения, если условие ложно.

```
const drink = personAge >= 18 ? "Wine" : "Juice";
```

Также можно составить цепочку из нескольких выражений:

```
const drink = personAge >= 18 ? "Wine" : personAge >= 6 ? "Juice" : "Milk";
```

Однако будьте осторожны, так как цепочка из нескольких выражений может привести к запутанному коду, который трудно поддерживать. Тернарный оператор особенно полезен в React, особенно в коде JSX, который принимает выражения только в фигурных скобках. Например, вы можете использовать троичный оператор для установки значения атрибута на основе определенного условия:

```
render(){
        const person = {
                name: 'Carlo',
                avatar: 'https://en.gravatar.com/...',
                description: 'Content Writer',
                theme: 'light'
        }

        return (
```

```
                <div
                        className='card'
                        style={
                                person.theme === 'dark' ?
                                 { background: 'black', color:
'white' } :
                                 { background: 'white', color:
'black'}
                        }>
                        <img
                                src={person.avatar}
                                alt={person.name}
                                style={ { width: '100%' } }
                        />
                         <div style={ { padding: '2px 16px' }
}>
                                <h3>{person.name}</h3>
                                <p>{person.description}.</p>
                        </div>
                </div>
        );
}
```

В приведенном выше коде мы проверяем условие person.theme === ‘dark’, чтобы установить значение атрибута style контейнера div.

Оценка короткого замыкания

Логический оператор AND (&&) оценивает операнды слева направо и возвращает true тогда и только тогда, когда все операнды истинны. Логический AND является оператором замыкания. Каждый операнд преобразуется в булево значение, и если результат преобразования оказывается ложным, оператор AND останавливается и возвращает исходное значение ложного операнда. Если все значения истинны, он возвращает исходное значение последнего операнда.

В JavaScript выражение true && всегда возвращает выражение, а выражение false && всегда возвращает false.

Оценка замыкания – это функция JavaScript, широко используемая в React, поскольку она позволяет выводить блоки кода на основе определенных условий. Вот пример:

```
{
        displayExcerpt &&
        post.excerpt.rendered && (
                <p>
                        <RawHTML>
                                { post.excerpt.rendered }
                        </RawHTML>
                </p>
        )

}
```

В приведенном выше коде, если displayExcerpt И post.excerpt.rendered оцениваются как true, React возвращает финальный блок JSX. Напомним, что “если условие истинно, то элемент сразу после && появится в выводе. Если условие ложно, React проигнорирует и пропустит его”.

Развернутый синтаксис

В JavaScript синтаксис распаковки позволяет развернуть итерируемый элемент, такой как массив или объект, в аргументы функции, литералы массива или литералы объекта. В следующем примере мы распаковываем массив в вызове функции:

```
function doSomething( x, y, z ){
         return `First: ${x} - Second: ${y} - Third: ${z} -
Sum: ${x+y+z}`;
}
const numbers = [3, 4, 7];

console.log( doSomething( ...numbers ) );
```

Синтаксис spread можно использовать для дублирования массива (даже многомерного) или для конкатенации массивов. В следующих примерах мы объединяем два массива двумя разными способами:

```
const firstArray = [1, 2, 3];
const secondArray = [4, 5, 6];
firstArray.push( ...secondArray );

console.log( firstArray );
```

В качестве альтернативы:

```
let firstArray = [1, 2, 3];
const secondArray = [4, 5, 6];
firstArray = [ ...firstArray, ...secondArray];

console.log( firstArray );
```

Для клонирования или слияния двух объектов можно также использовать синтаксис spread:

```
const firstObj = { id: '1', title: 'JS is awesome' };
const secondObj = { cat: 'React', description: 'React is easy'
};

// clone object
const thirdObj = { ...firstObj };

// merge objects
const fourthObj = { ...firstObj, ...secondObj }

console.log( { ...thirdObj } );
console.log( { ...fourthObj } );
```

Назначение деструктуризации

Еще одна синтаксическая структура, которую часто можно встретить в React, — это синтаксис деструктурирующего присваивания.

Синтаксис назначения деструктуризации позволяет вам распаковывать значения из массива или свойства объекта в отдельные переменные.

В следующем примере мы распаковываем значения из массива:

```
const user = ['Carlo', 'Content writer', 'Kinsta'];
const [name, description, company] = user;

console.log( `${name} is ${description} at ${company}` );
```

А вот простой пример деструктуризации присваивания с объектом:

```
const user = {
        name: 'Carlo',
        description: 'Content writer',
        company: 'Kinsta'
}
const { name, description, company } = user;

console.log( `${name} is ${description} at ${company}` );
```

Но мы можем сделать еще больше. В следующем примере мы распаковываем некоторые свойства объекта и присваиваем оставшиеся свойства другому объекту, используя синтаксис spread:

```
const user = {
        name: 'Carlo',
        family: 'Daniele',
        description: 'Content writer',
        company: 'Kinsta',
        power: 'swimming'
}
const { name, description, company, ...rest } = user;

console.log( rest ); // {family: 'Daniele', power: 'swimming'}
```

Вы также можете присваивать значения массиву:

```
const user = [];
const object = { name: 'Carlo', company: 'Kinsta' };
( { name: user[0], company: user[1] } = object );

console.log( user ); // (2) ['Carlo', 'Kinsta']
```

Обратите внимание, что круглые скобки вокруг оператора присваивания необходимы при использовании объектного

литерального деструктурирующего присваивания без объявления. Более подробный анализ деструктурирующего присваивания с несколькими примерами использования можно найти в веб-документации mdn.

Методы filter(), map() и reduce()

JavaScript предоставляет несколько полезных методов, которые часто используются в React.

filter()

Метод filter() создает неглубокую копию части заданного массива, отфильтрованную до элементов, удовлетворяющих условию заданной функции.

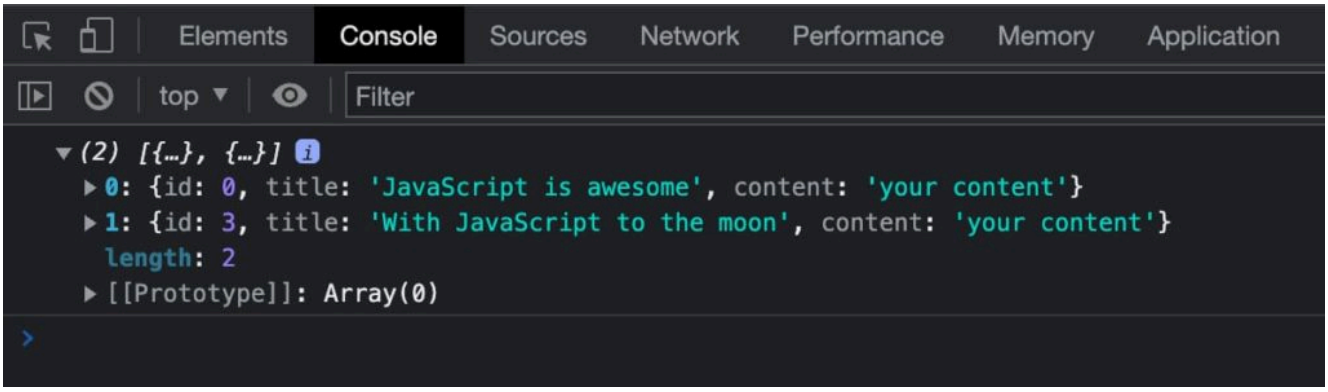
В следующем примере мы применяем фильтр к массиву numbers, чтобы получить массив, элементами которого являются числа больше 5:

```
const numbers = [2, 6, 8, 2, 5, 9, 23];
const result = numbers.filter( number => number > 5);

console.log(result); // (4) [6, 8, 9, 23]
```

В следующем примере мы получаем массив сообщений со словом ‘JavaScript’ в заголовке:

```
const posts = [
        {id: 0, title: 'JavaScript is awesome', content: 'your content'},
         {id: 1, title: 'WordPress is easy', content: 'your content'},
          {id: 2, title: 'React is cool', content: 'your content'},
        {id: 3, title: 'With JavaScript to the moon', content: 'your content'},
];

const jsPosts = posts.filter( post => post.title.includes(
```

```
'JavaScript' ) );

console.log( jsPosts );
```



Screenshot

map()

Метод map() выполняет предоставленную функцию над каждым элементом массива и возвращает новый массив, заполненный каждым элементом, полученным в результате выполнения функции обратного вызова.

```
const numbers = [2, 6, 8, 2, 5, 9, 23];
const result = numbers.map( number => number * 5 );

console.log(result); // (7) [10, 30, 40, 10, 25, 45, 115]
```

В компонентах React часто можно встретить метод map(), используемый для построения списков. В следующем примере мы отображаем объект WordPress posts для построения списка постов:

```
<ul>
        { posts && posts.map( ( post ) => {
                return (
                        <li key={ post.id }>
                                <h5>
                                        <a href={ post.link }>
                                                {
post.title.rendered ?
post.title.rendered :
                                                        __(
```

```
'Default title', 'author-plugin' )
                                                    }
                                                </a>
                                            </h5>
                                        </li>
                                    )
                            })}

</ul>
```

reduce()

Метод reduce() выполняет функцию обратного вызова (reducer) для каждого элемента массива и передает возвращаемое значение в следующую итерацию. Короче говоря, редуктор “сводит” все элементы массива к одному значению.

reduce() принимает два параметра:

- Функция обратного вызова, которая должна выполняться для каждого элемента массива. Она возвращает значение, которое при следующем вызове становится значением параметра accumulator. При последнем вызове функция возвращает значение, которое станет возвращаемым значением функции reduce().
- Начальное значение, которое является первым значением аккумулятора, передается в функцию обратного вызова.

Функция обратного вызова принимает несколько параметров:

- Аккумулятор: Значение, возвращенное при предыдущем вызове функции обратного вызова. При первом вызове оно устанавливается в начальное значение, если оно указано. В противном случае принимается значение первого элемента массива.
- Значение текущего элемента: Значение устанавливается на первый элемент массива (array[0]), если было задано начальное значение, иначе берется значение второго

элемента (array[1]).

- Текущий индекс – это индексная позиция текущего элемента.

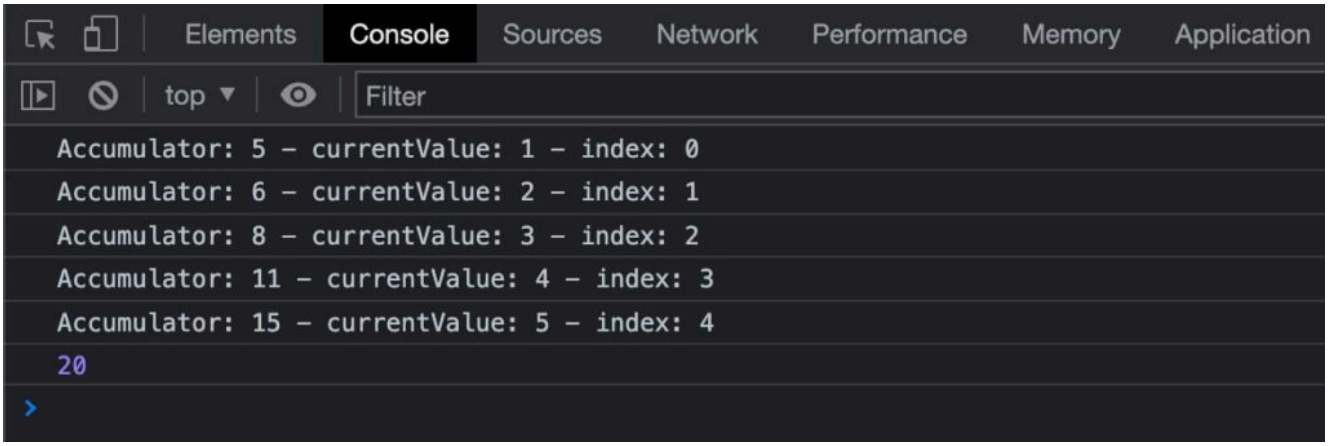
Пример сделает все более понятным.

```
const numbers = [1, 2, 3, 4, 5];
const initialValue = 0;
const sumElements = numbers.reduce(
          ( accumulator, currentValue ) => accumulator + currentValue,
        initialValue
);
console.log( numbers ); // (5) [1, 2, 3, 4, 5]

console.log( sumElements ); // 15
```

Давайте выясним подробнее, что происходит на каждой итерации. Вернитесь к предыдущему примеру и измените initialValue:

```
const numbers = [1, 2, 3, 4, 5];
const initialValue = 5;
const sumElements = numbers.reduce(
        ( accumulator, currentValue, index ) => {
                console.log('Accumulator: ' + accumulator + ' - currentValue: ' + currentValue + ' - index: ' + index);
                return accumulator + currentValue;
        },
        initialValue
);

console.log( sumElements );
```

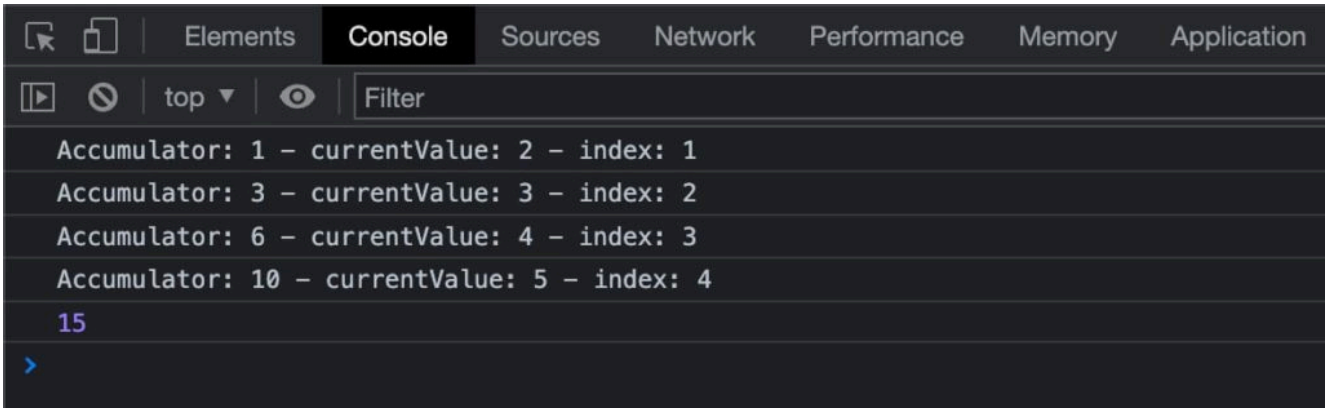
На следующем изображении показан вывод в консоли браузера:



Screenshot

Теперь давайте выясним, что происходит без параметра initialValue:

```
const numbers = [1, 2, 3, 4, 5];
const sumElements = numbers.reduce(
        ( accumulator, currentValue, index ) => {
                console.log( 'Accumulator: ' + accumulator + '
- currentValue: ' + currentValue + ' - index: ' + index );
                return accumulator + currentValue;
        }
);

console.log( sumElements );
```



Screenshot

Другие примеры и варианты использования обсуждаются на сайте mdn web docs.

Экспорт и импорт

Начиная с ECMAScript 2015 (ES6), появилась возможность

экспортировать значения из модуля JavaScript и импортировать их в другой сценарий. Вы будете широко использовать импорт и экспорт в своих приложениях React, поэтому важно хорошо понимать, как они работают.

Следующий код создает функциональный компонент. Первая строка импортирует библиотеку React:

```
import React from 'react';

function MyComponent() {
        const person = {
                name: 'Carlo',
                                                        avatar:
'https://en.gravatar.com/userimage/954861/fc68a728946aac04f853
1c3a8742ac22?size=original',
                description: 'Content Writer',
                theme: 'dark'
        }

        return (
                <div
                        className = 'card'
                        style = {
                                person.theme === 'dark' ?
                                 { background: 'black', color:
'white' } :
                                 { background: 'white', color:
'black'}
                        }>
                        <img
                                src = { person.avatar }
                                alt = { person.name }
                                style = { { width: '100%' } }
                        />
                        <div
                                  style = { { padding: '2px
16px' } }
                        >
                                <h3>{ person.name }</h3>
                                <p>{ person.description }.</p>
```

```
                        </div>
                </div>
        );
}

export default MyComponent;
```

Мы использовали ключевое слово import, за которым следует имя, которое мы хотим присвоить тому, что импортируем, а затем имя пакета, который мы хотим установить, как он указан в файле package.json.

Декларации импорта используются для импорта живых привязок только для чтения, экспортируемых другими модулями.

Обратите внимание, что в приведенной выше функции MyComponent() мы использовали некоторые возможности JavaScript, рассмотренные в предыдущих разделах. Мы включили значения свойств в фигурные скобки и присвоили значение свойства style, используя синтаксис условного оператора. Сценарий завершается экспортом нашего пользовательского компонента. Теперь, когда мы знаем немного больше об импорте и экспорте, давайте подробнее рассмотрим, как они работают.

Export

Объявление export используется для экспорта значений из модуля JavaScript.

Каждый модуль React может иметь два различных типа экспорта: именованный экспорт и экспорт по умолчанию.

Вы можете иметь несколько именованных экспортов для каждого модуля, но только один экспорт по умолчанию.

Например, вы можете экспортировать сразу несколько функций с помощью одного оператора export:

```
export { MyComponent, MyVariable };
```

Вы также можете экспортировать отдельные функции (function,

class, const, let):

```
export function MyComponent() { ... };

export let myVariable = x + y;
```

Но у вас может быть только один экспорт по умолчанию:

```
export default MyComponent;
```

Вы также можете использовать экспорт по умолчанию для отдельных функций:

```
export default function() { ... }

export default class { ... }
```

Import

После экспорта компонента его можно импортировать в другой файл, например, в файл index.js, вместе с другими модулями:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import MyComponent from './MyComponent';

const root = ReactDOM.createRoot( document.getElementById( 'root' ) );
root.render(
        <React.StrictMode>
                <MyComponent />
        </React.StrictMode>
);
```

В приведенном выше коде мы использовали декларацию import несколькими способами. В первых двух строках мы присвоили имя импортируемым ресурсам, в третьей строке мы не присвоили имя, а просто импортировали файл ./index.css. Последний оператор import импортирует файл ./MyComponent и присваивает имя. Давайте выясним разницу между этими импортами.

Всего существует четыре типа импорта:

Именованный импорт

```
import { MyFunction, MyVariable } from "./my-module";
```

Импорт по умолчанию

```
import MyComponent from "./MyComponent";
```

Импорт пространства имен

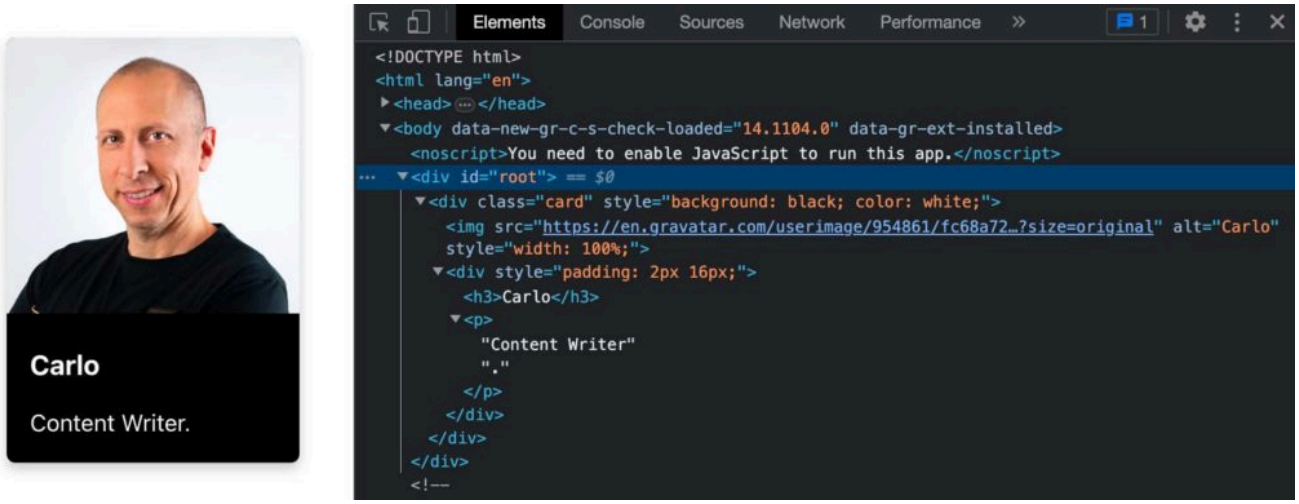
```
import * as name from "my-module";
```

Импорт побочных эффектов

```
import "module-name";
```

Оператор импорта без фигурных скобок используется для импорта экспорта по умолчанию. Оператор импорта с фигурными скобками используется для импорта именованного экспорта.

После добавления нескольких стилей в файл index.css ваша карточка должна выглядеть так, как показано на рисунке ниже, где вы также можете увидеть соответствующий HTML-код:



Screenshot

Обратите внимание, что декларации импорта можно использовать только в модулях верхнего уровня (не внутри функций, классов и т.д.).

Заключение

React – одна из самых популярных библиотек JavaScript на сегодняшний день и один из самых востребованных навыков в мире веб-разработки. С помощью React можно создавать динамические веб-приложения и продвинутые интерфейсы. Создание больших, динамичных и интерактивных приложений может быть легким благодаря многократно используемым компонентам. Но React – это библиотека JavaScript, и хорошее понимание основных возможностей JavaScript необходимо для начала вашего пути с React. Именно поэтому мы собрали в одном месте некоторые из функций JavaScript, которые чаще всего используются в React. Освоение этих функций даст вам преимущество в изучении React. А когда речь идет о веб-разработке, переход от JS/React к WordPress не требует особых усилий.