React введение

React — это JavaScript-библиотека для создания пользовательских интерфейсов. Обратите внимание, что это именно библиотека, а не фреймворк. React часто называют фреймворком, но это ошибка. Во-первых, его использование ни к чему вас не обязывает, не формирует «фрейм» проекта. Во-вторых, React выполняет единственную задачу: показывает на странице компонент интерфейса, синхронизируя его с данными приложения, и только этой библиотеки в общем случае недостаточно для того, чтобы полностью реализовать проект.

Вскоре после появления React и подобные ему решения (Vue.js, Svelte) практически захватили мир фронтенда: потому что они помогают решать проблемы, основываясь на идее декларативного программирования, а не на императивном подходе.

— Декларативный подход состоит в описании конечного результата (что мы хотим получить).

— При императивном подходе описываются конкретные шаги для достижения конечного результата (как мы хотим что-то получить).

Декларативный подход отлично подходит для создания интерфейсов.
Рассмотрим две версии простого приложения: на HTML и JS (императивный подход) и на React (декларативный подход). Программа будет показывать число и кнопку, и при нажатии на неё исходное число будет увеличиваться на единицу.

В рамках императивного подхода пошаговые инструкции для создания программы выглядят так:

1. объявляем начальные значения программы: присвоили константам ссылки на DOM-элементы, устанавливаем начальное значение счётчика
2. пишем обработчик increment, в котором мы увеличиваем текущее значение, и устанавливаем его в соответствующий элемент;
3. устанавливаем начальное значение счётчика (0);
4. устанавливаем обработчик для кнопки.

```
<!DOCTYPE html>
<html>
 <head>
   <title>Parcel Sandbox</title>
   <meta charset="UTF-8" />
 </head>

 <body>
   <div class="root">
     <h1 id="counter-value"></h1>
     <button id="increment-btn">+1</button>
   </div>

   <script>
     const counterValueElement = document.getElementById("counter-value");
     const incrementBtn = document.getElementById("increment-btn");
     let counterValue = 0;

     function increment() {
       counterValue += 1;
       counterValueElement.innerText = counterValue;
     }

     counterValueElement.innerText = counterValue;
     incrementBtn.addEventListener("click", increment);
   </script>
 </body>
</html>
```

Приложение на React

— Вызывая функцию React.useState, мы сообщаем React, что собираемся использовать какое-то изменяемое значение. В ответ React даёт нам само значение (value) и функцию, которая позволит его установить (setValue).
— Объявляем обработчик increment, который устанавливает новое значение счётчика с помощью вышеупомянутой функции.
— Создаём разметку приложения, используя синтаксис, похожий на HTML (на самом деле это JSX). Разметка повторяет то, что мы видели ранее, но теперь само значение счётчика и установка обработчика на клик находятся прямо в ней. Это как раз то место, где описывается конечный результат.
Весь код находится внутри функции App. В React она и другие похожие функции называются компонентами. Компонент — это фрагмент интерфейса, который содержит разметку и, при необходимости, связанную с ней логику. Все React-приложения строятся на компонентах. При этом сам компонентный подход появился задолго до React, но зде ью.

```
<!DOCTYPE html>
<html>
 <head>
   <title>Parcel Sandbox</title>
   <meta charset="UTF-8" />
 </head>

 <body>
   <div id="root"></div>

   <script
     src="https://unpkg.com/react@17/umd/react.development.js"
     crossorigin
   ></script>
   <script
     src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"
     crossorigin
   ></script>
   <script
     src="https://unpkg.com/@babel/standalone@7.13.6/babel.min.js"
     crossorigin
   ></script>
```

```
<script type="text/babel">
  function App() {
    const [value, setValue] = React.useState(0);

    function increment() {
      setValue(value + 1);
    }

    return (
      <div className="app">
        <h1>{value}</h1>
        <button onClick={increment}>+1</button>
      </div>
    );
  }

  ReactDOM.render(
    <div>
      <App />
    </div>,
    document.getElementById("root")
  );
</script>
</body>
</html>
```

В первом случае мы написали алгоритм для работы с элементами, значением и его изменения — шаги, необходимые для достижения результата.

Во втором, используя JSX-разметку и вспомогательные функции React, мы сразу описали результат, который хотим видеть. В этом и заключается отличие декларативного подхода от императивного.

Если сравнивать эти два приложения, то при использовании React можно выделить такие особенности:

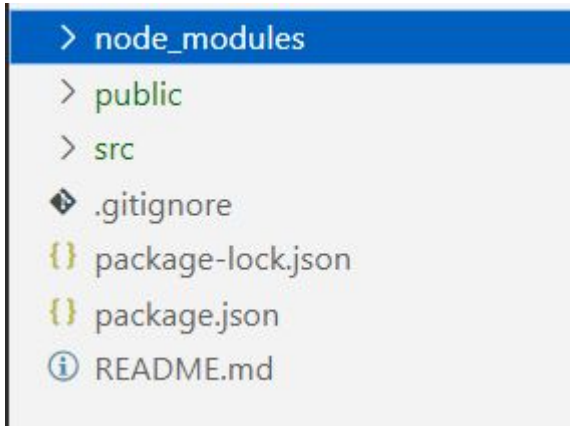
— Разметка и относящаяся к ней логика находятся рядом и связаны друг с другом. Это упрощает дальнейшую работу с кодом.
— Выделен счётчик с кнопкой в компонент. Это значит, что можно очень легко его переиспользовать: достаточно на 44 строке написать ещё один тег <App />, и на странице появятся уже два независимых друг от друга счётчика.
— Больше не нужно использовать идентификаторы для обращения к DOM-элементам, что также делает код более легко поддерживаемым.
— Состояние компонента изолировано: нет никакой возможности модифицировать значение извне, если явно такое не задумывали. Благодаря этому поток данных в приложении становится более предсказуемым, что упрощает разработку и отладку.

В React-приложениях мы не работаем напрямую с DOM-деревом. Вместо этого мы описываем разметку с помощью JSX, а React уже сам решает, как превратить её в реальные DOM-элементы. Это становится возможно благодаря абстракции, которая называется виртуальный DOM.

Установка фреймворка React

Для установки фреймворка понадобятся NodeJS не ниже версии 8.10. Для создания проекта в командной строке выполните следующие команды:

```
npx create-react-app my-app
cd my-app
npm start
```



Папка src - это будет ваша рабочая папка, в которой вы будете вести разработку вашего проекта.

В папке src найдите файл App.js - этот файл будет вашим основным рабочим файлом.

```
import React from 'react';

function App() {
  return <div>
    text
  </div>;
}

export default App;
```

для запуска достаточно перейти через терминал в папку **my-app** и выполнить следующую команду:

```
npm start
```

Введение в компонентный подход в React

Пусть есть сайт. На этом сайте можно выделить некоторые блоки: хедер, контент, сайдбар, футер и так далее. Каждый блок можно разделить на более мелкие подблоки. К примеру в хедере обычно можно выделить логотип, менюшку, блок контактов и так далее.
В React каждый такой блок называется компонентом. Каждый компонент может содержать в себе более мелкие компоненты, те в свою очередь еще более мелкие и так далее.
Каждому компоненту в React соответствует ES6 модуль, расположенный в папке src. Имя файла с модулем пишется с большой буквы и должно соответствовать функции, которая расположена в коде этого модуля. Например, файл с названием App.js должен содержать внутри себя функцию App:

```
import React from 'react';

function App() {
  return <div>
    text
  </div>;
}

export default App;
```

Один из компонентов должен быть основным - тем, к которому добавляются остальные компоненты. В React по умолчанию таким компонентом будет компонент App.

Макет сайта

В папке my-app/public в файле index.html расположен макет сайта. Вы можете размещать в нем любой HTML код - и вы увидите результат этого кода в браузере.

Кроме того, в макете сайта есть специальный див с id, равным root, в который монтируется основной компонент. Под монтированием понимается то, что в этот див будет выводится результат работы нашего компонента.

Результат работы компонента

В див с результатом будет выведено то, что возвращает через return функция компонента.

```
function App() {
  return <div>
    text
  </div>;
}
```

Обратите внимание на то, что div пишется без кавычек - просто пишем тег в JavaScript коде! Это основная особенность React.

На самом деле в React мы пишем не на языке JavaScript, а на языке JSX.

Процесс преобразования JSX в итоговый HTML код называется рендерингом.

Язык JSX - это обычный JavaScript, но с некоторыми дополнениями, позволяющими писать теги прямо в коде, без кавычек.

Теги можно возвращать через return:

```
function App() {
  return <div>
    text
  </div>;
}
```

Теги можно записывать в переменные или константы:

```
function App() {
  const  elem = <div>text</div>;
  return elem;
}
```

Все теги в JSX должны быть закрыты, в том числе теги, которые не требуют пары, например, input или br.

```
function App() {
  return <div>
    <input>
  </div>;
}
```

```
function App() {
  return <div>
    <input />
  </div>;
}
```

Верстка в JSX должна быть корректной. В частности, не все теги можно вкладывать друг в друга. Например, если в теге ul разместить абзац, это приведет к ошибке.

В теги можно добавлять атрибуты:

```
function App() {
  return <div id="elem">
    text
  </div>;
}
```

Некоторые атрибуты представляют собой исключения: вместо атрибута class следует писать атрибут className, а вместо атрибута for следует писать атрибут htmlFor:

```
function App() {
  return <div className="block">
    <label htmlFor="elem">text</label>
    <input id="elem" />
  </div>;
}
```

Практические задачи

Функция в вашем основном компоненте сейчас должна выглядеть следующим образом:

```
function App() {
  return <div>
    text
  </div>;
}
```

1. Поменяйте текст внутри дива. Посмотрите на изменения, произошедшие в браузере.
2. Добавьте в див несколько абзацев.
3. Добавьте в див несколько инпутов, разделенных тегами br.
4. Сделайте внутри дива список ul, содержащий в себе 10 тегов li.
5. Сделайте внутри дива таблицу с тремя рядами и тремя колонками.
6. Сделайте внутри дива три абзаца с различными CSS классами.

Внутри тега, который возвращается через return, может быть сколько угодно вложенных тегов:

```
function App() {
  return <div>
    <p>text1</p>
    <p>text2</p>
  </div>;
}
```

Открывающий тег обязательно должен быть написан на одной строке с командой return. Например, следующий код работать не будет:

```
function App() {
  return
    <div>
      <p>text1</p>
      <p>text2</p>
    </div>;
}
```

Для того, чтобы такой снос тега вниз заработал, наш тег необходимо взять в круглые скобки:

```
function App() {
  return (
    <div>
      <p>text1</p>
      <p>text2</p>
    </div>
  );
}
```

Через return нельзя возвращать сразу несколько тегов. Чтобы все заработало, нам придется взять наши теги в какой-нибудь общий тег:

```
function App() {
  return (
    <div>
      <div>
        <p>text1</p>
        <p>text2</p>
      </div>
      <div>
        <p>text3</p>
        <p>text4</p>
      </div>
    </div>
  );
}
```

В результате рендеринга мы получим дополнительный див, который мы в общем не хотели и ввели исключительно для корректности работы React. Этот див, к примеру, может сломать нам часть верстки.

Для избежания таких проблем в React введен специальный пустой тег <></>, который группирует теги, но в готовую верстку не попадает.

Возврат незакрытого тега

```
function App() {
  return <input />;
}
```

Возврат пустого тега

```
function App() {
  return <div></div>;
}
```

```
function App() {
  return <div />;
}
```

Исправьте ошибку кода

вернуть тег ul

```
function App() {
  return
    <ul>
      <li>text1</li>
      <li>text2</li>
      <li>text3</li>
    </ul>;
}
```

вернуть сразу два тега ul:

```
function App() {
  return <ul>
    <li>text1</li>
    <li>text2</li>
    <li>text3</li>
  </ul>
  <ul>
    <li>text1</li>
    <li>text2</li>
    <li>text3</li>
  </ul>;
}
```

вернуть инпут

```
function App() {
  return <input>;
}
```

вернуть три инпута

```
function App() {
  return <input><input><input>;
}
```

Вставка значений переменных и констант в JSX

```
function App() {
  const str = 'text';

  return <div>
    {str}
  </div>;
}
```

Кроме вставки константы в теге может быть еще какой-нибудь текст:

```
function App() {
  const str = 'text';

  return <div>
    eee {str} bbb
  </div>;
}
```

В один тег можно вставлять сколько угодно констант:

```
function App() {
  const str1 = 'text1';
  const str2 = 'text2';

  return <div>
    {str1} {str2}
  </div>;
}
```

Вставки констант также могут разделяться каким-либо текстом:

```
function App() {
  const str1 = 'text1';
  const str2 = 'text2';

  return <div>
    {str1} eee {str2}
  </div>;
}
```

Внутри фигурных скобок можно не только вставлять переменные и константы, но и выполнять произвольный JavaScript код.

```
function App() {
  const num1 = 1;
  const num2 = 2;

  return <div>
    {num1 + num2}
  </div>;
}
```

```
function App() {
  const num1 = 3;
  const num2 = 2;

  return <div>
    result: {num1 ** num2}
  </div>;
}
```

```
function App() {
  const name = 'john';
  const surname = 'smit';

  return <div>
    result: {name + ' ' + surname}
  </div>;
}
```

```
function App() {
  const num = 4;

  return <div>
    result: {Math.sqrt(num)}
  </div>;
}
```

Можно выполнять вставку не только примитивов, но также массивов и объектов.

```
function App() {
  const arr = [1, 2, 3];

  return <div>
    <p>{arr[0]}</p>
    <p>{arr[1]}</p>
    <p>{arr[2]}</p>
  </div>;
}
```

```
function App() {
  const obj = {a: 1, b: 2, c: 3};

  return <div>
    <p>{obj.a}</p>
    <p>{obj.b}</p>
    <p>{obj.c}</p>
  </div>;
}
```

Сделайте так, чтобы результатом рендеринга был тег ul, в тегах li которого будут стоять элементы массива.

```
function App() {
  const arr = [1, 2, 3, 4, 5];
}
```

Для значений имени и фамилии используйте значения элементов объекта.

```
function App() {
  const obj = {name: 'john', surname: 'smit'};
}

<p>
  name:    <span>john</span>, <br>
  surname: <span>smit</span>,
</p>
```

В переменных и константах можно хранить теги, выполняя затем их вставку в нужное место.

```
function App() {
  const str = <p>text</p>;

  return <div>
    {str}
  </div>;
}
```

результат рендеринга

```
<div>
  <p>text</p>
</div>
```

Несколько тегов, хранящихся в константе, обязательно нужно обернуть в какой-то общий тег.

```
function App() {
  const str = <p>text1</p><p>text2</p>;

  return <main>
    {str}
  </main>;
}
```

```
function App() {
  const str = <div><p>text1</p><p>text2</p></div>;

  return <main>
    {str}
  </main>;
}
```

Можно также использовать пустые теги

Теги, записываемые в константы, не обязательно писать на одной строке.

```
function App() {
  const str = <p>
    text
  </p>;

  return <div>
    {str}
  </div>;
}
```

```
function App() {
  const str = (
    <p>
      text
    </p>
  );

  return <div>
    {str}
  </div>;
}
```

Константы с тегами можно возвращать через return:

```
function App() {
  const str = <main>
    text
  </main>;

  return str;
}
```

Вставку переменных и констант можно делать не только в тексты тегов, но и в атрибуты. При этом кавычки от атрибутов не ставятся:

```
function App() {
  const str = 'elem';

  return <div id={str}>
    text
  </div>;
}
```

Результат работы

```
<div id="elem">
  text
</div>
```

вместо атрибута class следует писать атрибут className

```
function App() {
  const str = 'elem';

  return <div className={str}>
    text
  </div>;
}
```

вместо атрибута for следует писать атрибут htmlFor

```
function App() {
  const str = 'elem';

  return <div>
    <label htmlFor={str}>text</label>
  </div>;
}
```

Применение условий в JSX

в зависимости от содержимого константы show на экран вывелся или один, или другой текст

```
function App() {
  let text;
  const show = true;

  if (show) {
    text = 'text1';
  } else {
    text = 'text2';
  }

  return <div>
    {text}
  </div>;
}
```

```
function App() {
  let text;
  const show = true;

  if (show) {
    text = <p>text1</p>;
  } else {
    text = <p>text2</p>;
  }

  return <div>
    {text}
  </div>;
}
```

чтобы текст показывался, если в show будет true

```
function App() {
  let text;
  const show = true;

  if (show) {
    text = <p>text</p>;
  }

  return <div>
    {text}
  </div>;
}
```

возврат через return переменной, содержащей тег

```
function App() {
  let text;
  const show = true;

  if (show) {
    text = <p>text1</p>;
  } else {
    text = <p>text2</p>;
  }

  return text;
}
```

Практические задачи

1. Пусть в константе isAdult содержится true, если пользователю уже есть 18 лет, и false, если нет:

```
function App() {
  const isAdult = true;



}
```

Сделайте так, чтобы в зависимости от значения isAdult на экране показался или один абзац с текстом, или другой.

2. Пусть в константе isAdmin содержится true, если пользователь админ, и false, если нет:

```
function App() {
  const isAdmin = true;



}
```

Сделайте так, чтобы, если isAdmin имеет значение true, на экране показался див с абзацами. В противном случае ничего показывать не нужно.

Тернарный оператор

```
function App() {
  const show = true;

  return <div>
    {show ? 'text1' : 'text2'}
  </div>;
}
```

```
function App() {
  const show = true;

  return <div>
    {show ? <p>text1</p> : <p>text2</p>}
  </div>;
}
```

Оператор &&

```
function App() {
  const show = true;

  return <div>
    {show && <p>text</p>}
  </div>;
}
```

Инвертирование

```
function App() {
  const hide = false;

  return <div>
    {!hide && <p>text</p>}
  </div>;
}
```

Использование функций в React

```
function App() {
  function square(num) {
    return num ** 2;
  }

  function cube(num) {
    return num ** 3;
  }

  const sum = square(3) + cube(4);

  return <div>
    {sum}
  </div>
}
```

Вызов функций внутри тегов

```
function App() {
  function square(num) {
    return num ** 2;
  }

  return <div>
    {square(3)}
  </div>
}
```

```
function App() {
  function square(num) {
    return num ** 2;
  }

  function cube(num) {
    return num ** 3;
  }

  return <div>
    {square(3) + cube(4)}
  </div>
}
```

Навешивание событий в JSX

Сделаем так, чтобы по клику на блок выводился алерт с некоторым текстом.

```
function App() {
  function showMess() {
    alert('hello');
  }

  return <div>
    <button>show</button>
  </div>;
}
```

```
function App() {
  function showMess() {
    alert('hello');
  }

  return <div>
    <button onClick={showMess}>show</button>
  </div>;
}
```

Можно передать параметр при привязывании функции к событию. Для этого вызов функции следует обернуть в стрелочную функцию

```
function App() {
  function showMess(text) {
    alert(text);
  }

  return <div>
    <button onClick={() => showMess('user1')}>show1</button>
    <button onClick={() => showMess('user2')}>show2</button>
  </div>;
}
```

Объект Event в React

Внутри функции, привязанной к обработчику событий, доступен объект Event

```
function App() {
  function func(event) {
    console.log(event); // объект с событием
  }

  return <div>
    <button onClick={func}>act</button>
  </div>;
}
```

В переменную event попадает не родной объект Event браузера, а специальная кроссбраузерная обертка над ним со стороны React. Эта обертка называется SyntheticEvent. Эта обертка помогает событиям работать одинаково во всех браузерах. У нее такой же интерфейс, как и у нативного события, включая методы stopPropagation() и preventDefault().

Объект Event при передачи параметров

Пусть у нас есть некоторая функция func, которую мы хотим использовать в качестве обработчика события. Пусть эта функция принимает некоторый параметр:

```
function func(arg) {
  console.log(arg);
}
```

используем эту функцию в качестве обработчика, передав ей параметр:

```
function App() {
  function func(arg) {
    console.log(arg);
  }

  return <div>
    <button onClick={() => func('eee')}>act</button>
  </div>;
}
```

Пусть теперь кроме параметра мы хотим получить в нашей функции объект Event. Для этого нам нужно поступить следующим образом:

```
function App() {
  function func(arg, event) {
    console.log(arg, event);
  }

  return <div>
    <button onClick={event => func('eee', event)}>act</button>
  </div>;
}
```

Теги в массивах и циклах JSX

Пусть у нас в массиве хранятся теги:

```
function App() {
  const arr = [<p>1</p>, <p>2</p>, <p>3</p>];
}
```

Можно выполнить вставку содержимого нашей переменной с помощью фигурных скобок

```
function App() {
  const arr = [<p>1</p>, <p>2</p>, <p>3</p>];

  return <div>
    {arr}
  </div>;
}
```

Результат

```
<div>
  <p>1</p>
  <p>2</p>
  <p>3</p>
</div>
```

Массив с тегами можно создать в цикле:

```
function App() {
  const arr = [];

  for (let i = 0; i <= 9; i++) {
    arr.push(<p>{i}</p>);
  }

  return <div>
    {arr}
  </div>;
}
```

Формирование из массива с данными

Пусть у нас есть какой-нибудь массив с некими данными:

```
function App() {
  const arr = [1, 2, 3, 4, 5];
}
```

Положим каждый элемент этого массива в абзац и выведем эти абзацы в диве. Для этого можно воспользоваться любым удобным циклом JavaScript. Например, обычным for-of:

```
function App() {
  const arr = [1, 2, 3, 4, 5];
  const res = [];

  for (const elem of arr) {
    res.push(<p>{elem}</p>);
  }

  return <div>
    {res}
  </div>;
}
```

В React для таких дел более принято использовать цикл map

```
function App() {
  const arr = [1, 2, 3, 4, 5];

  const res = arr.map(function(item) {
    return <p>{item}</p>;
  });

  return <div>
    {res}
  </div>;
}
```

Проблема с ключами

В предыдущем примере мы формировали абзацы в цикле

```
const res = arr.map(function(item, index) {
  return <p>{item}</p>;
});
```

React требует, чтобы каждому тегу из цикла мы дали уникальный номер, чтобы React было проще с этими тегами работать в дальнейшем.
Этот номер добавляется с помощью атрибута key. В данном случае в качестве номера можно взять номер элемента в массиве. В нашем случае этот номер хранится в переменной index и значит исправленная строка будет выглядеть вот так:

```
function App() {
  const arr = [1, 2, 3, 4, 5];

  const res = arr.map(function(item, index) {
    return <p key={index}>{item}</p>;
  });

  return <div>
    {res}
  </div>;
}
```

Ключ key должен быть уникальным только внутри этого цикла, в другом цикле значения key могут совпадать со значениями из другого цикла.

Вывод массива объектов в JSX

```
function App() {
  const res = prods.map(function(item, index) {
    return <p key={index}>
      <span>{item.name}</span>:
      <span>{item.cost}</span>
    </p>;
  });

  return <div>
    {res}
  </div>;
}
```

Атрибут key мы добавляли порядковый номер элемента в массиве. На самом деле такая практика является плохой и ей следует пользоваться лишь в крайнем случае.
Дело в том, что при сортировке массива у элементов станут другие ключи и React не сможет правильно отслеживать связь между элементами массива и соответствующими тегами.
Более хорошей практикой будет добавить каждому продукту уникальный идентификатор, который и будет использоваться в качестве ключа.

```
const prods = [
  {id: 1, name: 'product1', cost: 100},
  {id: 2, name: 'product2', cost: 200},
  {id: 3, name: 'product3', cost: 300},
];
```

```
function App() {
  const res = prods.map(function(item) {
    return <p key={item.id}>
      <span>{item.name}</span>:
      <span>{item.cost}</span>
    </p>;
  });

  return <div>
    {res}
  </div>;
}
```

Вывод массива объектов в виде HTML таблицы

Пусть у нас дан наш массив с продуктами:

```
const prods = [
  {id: 1, name: 'product1', cost: 100},
  {id: 2, name: 'product2', cost: 200},
  {id: 3, name: 'product3', cost: 300},
];
```

выведем элементы нашего массива в виде HTML таблицы:

```
function App() {
  const rows = prods.map(function(item) {
    return <tr key={item.id}>
      <td>{item.name}</td>
      <td>{item.cost}</td>
    </tr>;
  });

  return <table>
    <tbody>
      {rows}
    </tbody>
  </table>;
}
```

Добавим заголовки колонок нашей

```
function App() {
  const rows = prods.map(function(item) {
    return <tr key={item.id}>
      <td>{item.name}</td>
      <td>{item.cost}</td>
    </tr>;
  });

  return <table>
    <thead>
      <tr>
        <td>название</td>
        <td>стоимость</td>
      </tr>
    </thead>
    <tbody>
      {rows}
    </tbody>
  </table>;
}
```

Задачи для решения

1. Сделайте так, чтобы метод render вывел на экран следующее:

```
<div>
    текст
</div>
```

2. Пусть в методе render есть переменная text с текстом 'текст'. С ее помощью выведите следующее:

```
<div>
    текст
</div>
```

3. Пусть в методе render есть переменная text с текстом ' текст '. С ее помощью выведите следующее

```
<div>
    <p>текст</p>
</div>
```

4. Пусть в методе render есть переменная text1 с текстом ' текст1 ' и переменная text2 с текстом ' текст2 '. С их помощью выведите следующее:

```
<div>
    <p>текст1</p>
    <p>!!!</p>
    <p>текст2</p>
</div>
```

5. Пусть в методе render есть переменная attr с текстом 'block'. Сделайте так, чтобы метод render вывел на экран следующее (значение атрибута id должно вставится из переменной attr):

```
<div id="block">
    текст
</div>
```

6. Пусть в методе render есть переменная str с текстом 'block'. Сделайте так, чтобы метод render вывел на экран следующее (значение атрибута class должно вставится из переменной str):

```
<div class="block">
    текст
</div>
```

7. Дан див с текстом. Установите этому диву зеленый цвет, границу и border-radius в 30px.

8. Пусть в методе render есть переменная show, которая может иметь значение true или false. Сделайте так, чтобы, если эта переменная равна true, метод render вывел на экран следующее:

```
<div>
    текст 1
</div>
```

А если эта переменная равна false, то следующее:

```
<div>
    текст 2
</div>
```

9. Пусть в методе render есть переменная arr, в которой лежит массив с элементами ['a', 'b', 'c', 'd', 'e']. Сделайте так, чтобы метод render вывел на экран следующее (в каждую лишку запишется один из элементов массива):

```
<ul>
    <li>a</li>
    <li>b</li>
    <li>c</li>
    <li>d</li>
    <li>e</li>
</ul>
```

10. Пусть класс App имеет метод getText(), который своим результатом возвращает ' текст '. Используя метод getText() в методе render выведите на экран следующее:

```
<div>
    <p>текст</p>
</div>
```

11.Пусть класс App имеет метод getNum1(), который своим результатом возвращает число 1 и метод getNum2(), который своим результатом возвращает число 2. Используя эти методы в методе render выведите на экран сумму результатов этих методов (3 - результат сложения getNum1() и getNum2()):

```
<div>
    текст 3
</div>
```

12. Дан массив с работниками. У каждого работника есть имя, фамилия, количество отработанных дней и зарплатная ставка за день. Выведите этих работников на экран в виде таблицы. Сделайте так, чтобы в последней колонке автоматически рассчитывалась зарплата работника (количество отработанных дней умножить на ставку). Под таблицей также выведите суммарную зарплату всех работников.