Разбор задач отборочного тура трека

Frontend

Вопрос 1

Дано SVG-изображение размером 500 × 200. Требуется, чтобы в приложении оно отображалось размером 60 × 25. Выберите правильный набор атрибутов SVG для масштабирования.

- `width="60" height="25" viewBox="0 0 500 200"`
- `width="60" height="25" viewBox="500 0 200 0"`
- `width="60" height="25" viewBox="500 200"`
- `width="500" height="200" viewBox="60 25 0 0"`
- `width="500" height="200" viewBox="60 0 25 0"`
- `width="500" height="200" viewBox="60 25"`

ПРАВИЛЬНЫЙ ОТВЕТ

`width="60" height="25" viewBox="0 0 500 200"`

РЕШЕНИЕ

Атрибуты `width` и `height` отвечают за размер элемента (контейнера картинки) на странице.

С помощью `viewBox` можно масштабировать и позиционировать SVG-изображение относительно его контейнера. Задается `viewBox` с помощью четырех последовательных значений: `min-x`, `min-y`, `width`, `height`.

`min-x` и `min-y` задают начало пользовательской системы координат, `width` и `height` определяют ширину и высоту «пользовательской области просмотра» и одновременно отвечают за масштабирование изображения.

Так как в приложении изображение должно быть 60 × 25, это значения для атрибутов `width` и `height`. Чтобы картинка корректно масштабировалась под этот размер, нужно указать, что в «область просмотра» `viewBox` попадает все изображение, то есть 500 × 200.

Вопрос 2

У элемента заданы `width` и `height`, но если добавляем `padding`, то ширина и высота увеличиваются. Какое свойство помогает этого избежать?

- `box-sizing: content-box;`
- `box-sizing: border-box;`
- `box-sizing: initial;`

ПРАВИЛЬНЫЙ ОТВЕТ

Как следует из документации, по умолчанию в блоковой модели CSS свойства `width` и `height` задают размеры контента элемента и не учитывают границы и отступы. Такое поведение соответствует свойству `box-sizing: content-box`.

А вот `box-sizing: border-box` говорит браузеру учитывать границы и внутренние отступы в значениях, указанных для ширины и высоты.

Таким образом, правильный ответ — `box-sizing: border-box`.

Вопрос 3

Даны стили и верстка:

```
<style>
      .wrapper {
          height: 40px;
      }
     .wrapper div {
         width: 70px;
         height: 40px;
     }
     .z2 {
         z-index: 2;
         margin-bottom: -40px;
         border-right: 20px solid white;
         background: green;
     }
     .z1 {
         z-index: 1;
         margin-left: 70px;
         border-left: 20px solid black;
         background: red;
     }
</style>

<div class="wrapper">
     <div class="z2"></div>
     <div class="z1"></div>
</div>
```

В каком из этих вариантов отображение будет отличаться от других?

- Добавить `z2 position: static`, `z1 position: static`
- Добавить `z2 position: relative`, `z1 position: static`
- Добавить `z2 position: relative`, `z1 position: relative`
- Добавить `z2 position: absolute`, `z1 position: relative`

ПРАВИЛЬНЫЙ ОТВЕТ

Добавить `z2 position: static`, `z1 position: static`

РЕШЕНИЕ

Задача основана на особенности элементов со свойством `position: static` — на них, в отличие от позиционированных элементов, не влияет свойство `z-index`.

Если попробовать воспроизвести все варианты ответов, кроме правильного, то между блоками вы увидите белую полоску, потому что элемент с классом `z2` оказывается поверх элемента с `z1` за счет наличия у него `z-index: 2`.

Если же в стили класса `z2` добавить `position: static`, то `z-index` перестает действовать, элемент оказывается внизу и вы видите черную полоску от `z1`.

Вопрос 4

Дана верстка:

```
<div class="container">
    <div class="div2 left-section">2</div>
    <div class="div3 left-section">3</div>

    <div class="div4 right-section">4</div>
    <div class="div1 right-section">1</div>
    <div class="div5 right-section">5</div>
</div>
```

И фрагмент стилей:

```
.container { width: 50em; }

.left-section { width: calc(100% - 10em); }

.right-section { width: 10em; }

.div1 { height: 7em; background-color: red; }

.div2 { height: 5em; background-color: orange; }

.div3 { height: 7em; background-color: yellow; }

.div4 { height: 5em; background-color: green; }

.div5 { height: 10em; background-color: aqua; }
```

Какие из наборов стилей дают такое отображение?

Вариант 1

```
.container {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  min-height: 12em;
}
```

Вариант 2

```
.div3:after {
  height: 10em;
}

.container {
  display: flex;
  flex-direction: column;
```

```
  flex-wrap: wrap;
  height: 22em;
}
```

Вариант 3

```
.div3 {
  margin-bottom: 7em;
}

.container {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  height: 22em;
}
```

Вариант 4

```
.container {
  display: grid;
  grid-template-areas:
    "2 4"
    "3 1"
    ". 5";
}

.div1 { grid-area: 1; }

.div2 { grid-area: 2; }
```

```
.div3 { grid-area: 3; }

.div4 { grid-area: 4; }

.div5 { grid-area: 5; }
```

ПРАВИЛЬНЫЙ ОТВЕТ — ВАРИАНТ 3

РЕШЕНИЕ

Вариант 1

```
.container {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  min-height: 12em;
}
```

Несмотря на то что добавлен перенос элементов, контейнер никак не ограничен по максимальной высоте, а значит, элементы останутся в одной колонке:

Вариант 2

```
.div3:after {
  height: 10em;
}
```

```
.container {
  display: flex;
  flex-direction: column;
  flex-wrap: wrap;
  height: 22em;
}
```

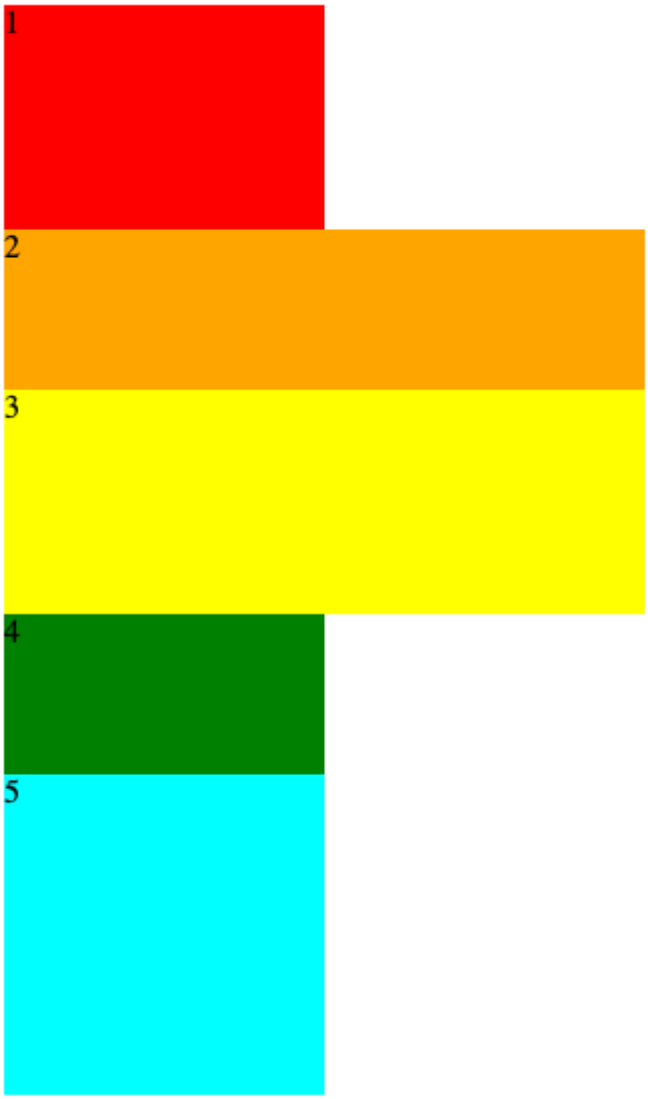
Наличие стиля с `:after` никак не влияет на высоту блока 3, ее недостаточно для того, чтобы суммарная высота блоков 2 и 3 превысила ограничение `22em`, из-за чего блок 4 остается слева:

Вариант 4

```
.container {
  display: grid;
  grid-template-areas:
    "2 4"
    "3 1"
    ". 5";
}
```

```
.div1 { grid-area: 1; }

.div2 { grid-area: 2; }

.div3 { grid-area: 3; }

.div4 { grid-area: 4; }

.div5 { grid-area: 5; }
```

Если задать свойству `grid-area` только одно число, то произойдет задание строки грида, на которой будет располагаться элемент, а не именование области. Таким образом получится столбец «отсортированных» элементов:



В случае правильного варианта 3 за счет наличия нижнего отступа у блока 3 произойдет выталкивание блока 4 на следующий столбец, в результате чего будет получено требуемое отображение.

Вопрос 5

Какие элементы из приведенного списка являются inline?

- label
- div
- header
- span
- p
- a

ПРАВИЛЬНЫЙ ОТВЕТ

Из приведенных в задании элементов inline-элементами являются `label`, `span` и `a`. Остальные элементы — блочные.

Вопрос 6

Выберите варианты кода, с помощью которого можно предотвратить перезагрузку страницы после отправки формы.

Вариант 1

```
<form onsubmit="return false;" method="post">
  <input type="text" />
  <button type="submit">Отправить</button>
</form>
```

Вариант 2

```
<form id="form" method="post">
  <input type="text" />
  <button type="submit">Отправить</button>
</form>

const form = document.getElementById('form');

form.addEventListener('submit', (e) => {
  e.stopReload();
});
```

Вариант 3

```
<form id="form" method="post">
  <input type="text" />
  <button type="submit">Отправить</button>
</form>
```

```
const form = document.getElementById('form');

form.addEventListener('submit', (e) => {
  e.preventDefault();
});
```

ПРАВИЛЬНЫЕ ОТВЕТЫ — 1 И 3

Во втором примере мы просто перехватываем событие отправки формы и отменяем его.

Первый пример требует знаний об алгоритме работы инлайн-обработчиков событий. Если сделать упрощение из спецификации, то значение `onsubmit` будет передано в конструктор `Function`. Новая получившаяся функция будет назначена как обработчик события `submit`.

Это можно представить следующим псевдокодом:

```
<form>.onsubmit = new Function('event', 'return false').
```

Также согласно спецификации (шаг 5) возвращаемое значение из обработчика будет отменять отправку формы в случае, если обработчик вернет `false`.

Вопрос 7

На элемент повесили два обработчика. Необходимо при клике на элемент блокировать вызов второго обработчика. Как вы это сделаете?

- `event.stopPropagation();`
- `event.preventDefault();`
- `event.stopImmediatePropagation();`

ПРАВИЛЬНЫЙ ОТВЕТ

```
event.stopImmediatePropagation();
```

Различия между этими методами хорошо объяснены [в статье на MDN.]

Вопрос 8

На странице подключены два скрипта:

```
<script async src="test-1.js"></script>
<script defer src="test-2.js"></script>
```

Почему-то они выполняются в разном порядке: иногда раньше выполняется первый скрипт, а иногда второй.

Выберите вариант ответа, который позволит гарантировать корректный порядок выполнения скриптов, не блокируя рендеринг страницы.

Вариант 1

```
<script src="test-1.js"></script>
<script src="test-2.js"></script>
```

Вариант 2

```
<script async src="test-1.js"></script>
<script async src="test-2.js"></script>
```

Вариант 3

```
<script defer src="test-1.js"></script>
<script defer src="test-2.js"></script>
```

ПРАВИЛЬНЫЙ ОТВЕТ — 3

В отличие от async, атрибут defer гарантирует порядок исполнения скриптов. Более детальное описание можно найти в спецификации и статье на MDN.

Вопрос 9

Выберите правильный порядок рендеринга страницы.

- DOM → Внешние ресурсы (link, script) → CCSOM → Выполнение JavaScript → Слияние DOM и CCSOM в Render Tree → Расчет макета и отрисовка результата
- Внешние ресурсы (link, script) → CCSOM → Выполнение JavaScript → DOM → Слияние DOM и CCSOM в Render Tree → Расчет макета и отрисовка результата
- Внешние ресурсы (link, script) → Выполнение JavaScript → DOM → CCSOM → Слияние DOM и CCSOM в Render Tree → Расчет макета и отрисовка результата

ПРАВИЛЬНЫЙ ОТВЕТ — 1

Информацию по рендерингу веб-страниц можно получить на официальном ресурсе MDN и в цикле статей на web.dev. Также есть замечательная статья на русском языке, раскладывающая все по полочкам.

Вопрос 10

Перед вами функция `getTop3`, возвращающая результаты топ-3 участников соревнования.

Функция работает не так, как ожидается. Найдите и исправьте ошибку.

```
getTop3([1, 10, 5, 1, 12, 8]); // expected 12, 10, 8
```

КОД ИЗ ЗАДАНИЯ

```
function getTop3(score) {
   return score.sort().reverse().slice(0, 3);
}
```

РЕШЕНИЕ

С десятого вопроса начался блок задач на написание кода. Для разминки мы выбрали два задания на исправление ошибок, с которыми сталкивалось большинство javascript-разработчиков.

В этой задаче надо было вспомнить, что sort по умолчанию сортирует элементы массива как строки. Поэтому массив `[1, 10, 5, 1, 12, 8]` после сортировки превращается в `[1, 1, 10, 12, 5, 8]`. Чтобы отсортировать элементы как числа, надо передать функцию-компаратор:

```
score.sort((a, b) => a - b
```

Также заметим, что элементы массива можно сразу отсортировать по убыванию, тогда reverse не понадобится.

```
function getTop3(score) {
   return score.sort((a, b) => b - a).slice(0, 3);
}
```

Вопрос 11

Разберитесь, что делает функция `replaceCurrencyNameWithSymbol`, и исправьте ошибку.

```
function replaceCurrencyNameWithSymbol(str, currencyName, currencySymbol) {
    return str.replace(currencyName, currencySymbol);
}

// expected 'Приход: 10 ₽, сумма: 100 ₽'
replaceCurrencyNameWithSymbol('Приход: 10 руб., сумма: 100 руб.', 'руб.', '₽');
```

РЕШЕНИЕ

Еще одна задача на исправление частой ошибки. Здесь дело в том, что `replace` заменяет только первое вхождение `currencyName`.

Чтобы починить тесты, можно воспользоваться функцией `replaceAll`. Мы запускали тесты на свежей версии Node.js, в которой `replaceAll` уже поддерживается. Это самое короткое верное решение — достаточно добавить три буквы к исходному коду:

```
function replaceCurrencyNameWithSymbol(str, currencyName, currencySymbol) {
   return str.replaceAll(currencyName, currencySymbol);
}
```

Альтернативное решение — через RegExp:

```
function replaceCurrencyNameWithSymbol(str, currencyName,
currencySymbol) {
   const regexp = new RegExp(currencyName, 'g');
   return str.replace(regexp, currencySymbol);
}
```

Кстати, решение через RegExp написал каждый пятый участник. А еще 3% отправок, прошедших тесты, разбивали строку функцией split и соединяли функцией join, используя в качестве сепаратора `currencySymbol`, либо вызывали replace в цикле.

Вопрос 12

Реализуйте функцию `isNotNull`, чтобы приведенный код работал корректно и не было ошибок типизации.

```
function mapAndFilterNulls(someData: string[]): number[] {
    return
someData.map(someMappingFunction).filter(isNotNull);
}

function someMappingFunction(el: string): number | null {
    return isNaN(+el) ? null : +el;
}
```

РЕШЕНИЕ

Чтобы избавиться от ошибки, нужно использовать Type Guard для сужения диапазона типов, например:

```
function isNotNull<T>(el: T | null): el is T {
   return el !== null;
}
```

Вопрос 13

Выберите, какие действия с объектом приведут к такому выводу в консоль:

```
const object = {name: 'Fred', surname: 'Mercury'};
// ???
console.log({...object}); // -> {surname: Mercury}
```

Вариант 1

```
Object.defineProperty(object, 'name', {
  enumerable: true
});
```

Вариант 2

```
Object.defineProperty(object, 'name', {
  enumerable: false
});
```

Вариант 3

```
delete object.name;
```

Вариант 4

```
object = {surname: 'Mercury'};
```

ПРАВИЛЬНЫЕ ОТВЕТЫ — 2 И 3

Чтобы справиться с этой задачей, необходимо знать о существовании дескрипторов свойств в JavaScript, о способах их изменения, а также о том, как работает spread-оператор. В нашем случае достаточно изменить дескриптор `enumerable` свойства `name` на `false`. И теперь, когда spread-оператор будет перечислять свойства объекта, он пропустит свойства с пометкой `enumerable: false`.

Ответ 4 не подходит, так как `object` объявлен как `const`.

Вопрос 14

Допишите код так, чтобы в консоль выводилось `true`:

```
const object = {};
// TODO: здесь ваш код
console.log(object == '1');
```

РЕШЕНИЕ

Первая часть более сложного вопроса 15 предполагает знание алгоритма приведения объекта к строке. Есть несколько вариантов решения с переопределением `Symbol.toPrimitive`, `valueOf()` и `toString()`.

```
const object = {};

object[Symbol.toPrimitive] = () => '1';

console.log(object == '1');
```

Вопрос 15

Допишите код так, чтобы в консоль выводилось `true`:

```
const object = Object.freeze({});
// TODO: здесь ваш код
console.log(object == '1');
```

РЕШЕНИЕ

Усложнение вопроса 14 предполагает знание алгоритма получения свойств по цепочке прототипов. Так как Object.freeze теперь запрещает нам модифицировать объект напрямую и модификатор const у переменной не позволяет переопределить ссылку на объект, мы обращаемся к прототипу object и получаем аналогичное предыдущему вопросу решение:

```
const object = Object.freeze({});

object.__proto__.valueOf = () => '1';

console.log(object == '1');
```

Вопрос 16

Реализуйте тип — неотрицательное число для работы в функции квадратного корня.

```
function sqrt<N extends number>(n: NonNegativeNumber<N>):
number {
    return Math.sqrt(n);
}
```

РЕШЕНИЕ

```
type NonNegativeNumber<N extends number> = number extends N
? never
: `${N}` extends `-${string}`
? never
: N;
```

То есть такой тип сводится к двум условиям:

```
number extends N
 ? never
 : …
```

Первое условие гарантирует, что в наш тип-кандидат не входит множество всех возможных чисел. Такое множество наполовину состоит из отрицательных чисел, а их мы отметаем.

```
`${N}` extends `-${string}`
? never
: N;
```

Во втором условии нам очень помогают TypeScript-литералы. Если наше число в виде строки начинается со знака минус, то это число отрицательное — такие мы исключаем (сбрасываем тип к *never*). Иначе — возвращаем суженное до неотрицательных число.

Вопрос 17

Напишите тип для массива, элементами которого могут быть строка или такой же массив (элементами которого могут быть строка или такой же массив и т. д.)

Например, тип должен позволять такое:

```
const myRecursiveArray = ['test', [], ['test2']];
```

РЕШЕНИЕ

```
type RecursiveArray = Array<RecursiveArray | string>;
```

Имплементация не содержит сюрпризов. Мы буквально декларируем тип массива, состоящего из строки или такого же массива. Поскольку TS позволяет делать рекурсивные типы, проблем с работой этого типа у нас не возникнет и такой массив будет поддерживать любой уровень вложенности.

Вопрос 18

Перед вами реализация структуры данных — множества, в которое элементы можно добавлять только один раз. После удаления элемент нельзя добавить снова.

Подумайте, что здесь не так, и исправьте ошибку.

```
class OnceSet extends Set {

    added = new Set();


    add(el) {
        if (this.added.has(el)) {
            return this;
        }

        this.added.add(el);
        return super.add(el);
    }
}
```

РЕШЕНИЕ

При попытке создать экземпляр класса `OnceSet` путем передачи массива значений в конструктор будет возникать ошибка “`cannot read properties of undefined (reading 'has')`”.

При попытке дебага действительно можно увидеть, что `this.added` на момент вызова конструктора еще не инициализирован. Но как же получается, что метод `add` вызывается раньше инициализации свойства у экземпляра класса?

Ответ кроется в механизме работы наследования и тонкостей реализации структуры данных `Set`, а именно [конструктора.]

Как видно из документации, при попытке вызова конструктора с аргументом типа `iterable` будет вызван метод `add` на каждый элемент этого `iterable`. Действительно, можно рассмотреть реализацию без использования синтаксического сахара для инициализации поля класса, тогда проблема будет более очевидной:

```
class OnceSet extends Set {
    constructor(...args) {
        super(...args);
        this.added = new Set();
    }
    add(el) {
        if (this.added.has(el)) {
            return this;
        }

        this.added.add(el);
        return super.add(el);
    }
}
```

Существует несколько подходов для исправления этой ошибки. Можно, например, инициализировать поле `added` непосредственно в методе `add`.

```
class OnceSet extends Set {

    add(el) {

        if (!this.added) {

            this.added = new Set();

        }

        if (this.added.has(el)) {

            return this;

        }


        this.added.add(el);

        return super.add(el);

    }

}
```

Вопрос 19

Напишите функцию `requestWithRetry`, которая принимает функцию запроса и количество раз, которое этот запрос нужно повторять до успешного результата.

Retry должен вернуть промис с результатом или реджектом в случае, если все попытки провалились.

Пример использования:

```
const results = await requestWithRetry(someFunction, 5);
```

РЕШЕНИЕ

Два основных пути решения этой задачи — это цикл и рекурсия. А основа современного лаконичного решения — async/await-синтаксис.

```
async function requestWithRetry(request, attempts = 1) {
    let response;
    for (let attempt = 0; attempt < attempts; attempt++) {
        try {
            response = await request();
            return response;
        } catch {}
    }
```

```
    throw new Error("Error");
}
```

В этом решении мы просто делаем перебор попыток от первой до последней, повторяя запрос. Получим результат — сразу вернем его. Не получим за все попытки — бросаем ошибку.

Вопрос 20

Напишите функцию, которая принимает функцию запроса и максимальное время его ожидания. Возвращает промис с результатом запроса или реджектом в случае, если ответ не получен за назначенное время.

РЕШЕНИЕ

К этой задаче тоже можно подойти через разные возможности JS: можно решить «в лоб», через создание нового промиса с запросом и таймаутом, а можно воспользоваться методом `Promise.race`, который вернет результат победителя гонки между ними.

```
function requestWithTimeout(request, timeout) {
   return Promise.race([
       request(),
       new Promise((_, reject) =>
           setTimeout(
               () => reject(new Error(`Request timed out after
${timeout}ms`)),
               timeout
           )
       ),
   ]);
}
```

Вопрос 21

Реализуйте функцию `excludePaths`, принимающую объект и список путей, которые надо исключить, и возвращающую новый объект без этих путей. Например:

```
const alice = {
    name: 'Alice',
    age: 20,
    track: {
        title: 'Frontend',
        score: 100
    }
};

const newObj = excludePaths(alice, ['age', 'track.score']);

console.log(newObj); // -> { name: 'Alice', 'track': {title:
'Frontend'}}
```

РЕШЕНИЕ

Для решения этой задачи удобнее всего написать рекурсивную функцию удаления. Разбиваем переданные для удаления пути по точкам и пытаемся рекурсивно удалять. Не забываем создать копию исходного объекта.

```
function excludePaths(obj, paths) {

  const copy = JSON.parse(JSON.stringify(obj));
```

```
  paths.forEach(path => {
      const split = path.split('.');
      excludePath(copy, split, 0);
  });

  return copy;
}

function excludePath(obj, path, index) {
  if (index + 1 === path.length) {
      delete obj[path[index]];
      return;
  }
   if (path[index] in obj) {
      excludePath(obj[path[index]], path, index + 1);
  }
}
```

Вопрос 22

От коллег вам достался код, возвращающий N-ю страницу рейтинга участников.

Вам показалось, что код избыточный. Перепишите его более лаконично, не меняя функциональность:

```
// page 1-based
function getParticipants(arr, track, size, page) {
   const arr2 = [];
    for (let i = 0; i < arr.length; i++) {
     if (arr[i].track === track) {
       arr2.push(arr[i]);
     }
   }
    for (let i = 0; i < arr2.length; i++) {
     for (let j = i + 1; j < arr2.length; j++) {
       if (arr2[i].name > arr2[j].name) {
         const temp = arr2[i];
         arr2[i] = arr2[j];
         arr2[j] = temp;
       }
     }
   }

   const arr3 = [];

   const first = size * (page - 1);
   for (let i = 0; i < arr2.length; i++) {
       if (first <= i && i < first + size) {
           arr3.push(arr2[i]);
       }
```

```
    }

    return arr3;
  }
```

РЕШЕНИЕ

Здесь нам помогут стандартные функции JavaScript. Заметим, что сначала элементы массива `arr` фильтруют по полю track. Это можно сделать функцией `filter`. Затем происходит сортировка по имени, причем за квадратичное время. Лучше воспользоваться функцией `sort`. И наконец, от полученного массива отрезаются элементы нужной страницы. Для этого подойдет, например, функция `slice`.

Итого — все то же самое можно сделать в четыре строки:

```
function getParticipants(arr, track, size, page) {
  const first = size * (page - 1);

  return arr.filter(user => user.track === track)
      .sort((a, b) => a.name.localeCompare(b.name))
      .slice(first, first + size);
}
```

IT's TINKOFF

Вопрос 23

Напишите типизированную функцию `getMax`, которая принимает на вход массив элементов и функцию сравнения, а возвращает самый большой элемент массива согласно функции сравнения.

Если на вход передан пустой массив, то функция должна возвращать `undefined`.

РЕШЕНИЕ

Эту задачу можно решать разными путями, но суть оптимального решения одна: мы перебираем массив целиком один раз, на каждой итерации поддерживая текущий максимум. Два основных решения с JS — классическим циклом for или через reduce:

```
function getMax<T>(arr: Array<T>, comparator: (a: T, b: T) => number): T |
undefined {
    if (arr.length === 0) {
        return undefined;
    }

    let max = arr[0];
    for (let i = 1; i < arr.length; i++) {
        if (comparator(arr[i], max) > 0) {
            max = arr[i];
        }
    }
    return max;
```

```
}
```

Для типизации используем Generic в TypeScript, чтобы обозначить, что массив, компаратор и возвращаемое значение имеют общий тип.

Вопрос 24

Вам надо загрузить огромный файл на сервер. Напишите функцию, которая сделает это чанками в несколько потоков.

РЕШЕНИЕ

Часто в работе frontend-приложения требуется загрузить файл на сервер. Иногда этот файл очень большой, иногда есть ограничения на максимальный размер запроса. И в таких случаях приходится разбивать файл на куски (чанки), которые отправляются на сервер последовательно. В ряде случаев мы можем отправлять по несколько чанков параллельно.

Так и появилась задача — загрузки файла в несколько потоков.

Контракт в задаче был задан:

```
/** Настройки загрузки */
type Options = {
  readonly maxChunks: number;
  readonly chunkSize: number;
}

/** Интерфейс источника данных */
interface Source {
  readonly size: number;
```

```
  read(start: number, end: number): Blob;
}

/** Функция обратного вызова, отправляющая данные на сервер */
type SendCb = (data: Blob, offset: number) => Promise<void>;
```

Определимся с алгоритмом работы загрузчика. Мы создадим массив «потоков», размер которого будет равен максимальному количеству параллельно отправляемых чанков. Под потоком будем подразумевать промис, который будет последовательно считывать и отправлять чанки друг за другом, пока они не кончатся. И с помощью `Promise.all` будем ждать завершения всех потоков.

Для этого внутри функции `upload` объявим пустую функцию `uploadNextChunk` (функцию потока):

```
function uploadNextChunk(): Promise<void> {
    ...
}
```

Тогда сам код функции загрузки будет выглядеть как:

```
await Promise.all(
    new Array(options.maxChunks).fill(0).map(
        () => uploadNextChunk(),
    ),
```

```
);
```

Конструкция `new Array(<количество>).fill(0)` нужна для того, чтобы созданный массив имел значения. Если бы мы использовали только `new Array(<количество>)`, то не смогли бы воспользоваться методом `map`.

Чтобы все части загрузились по одному разу, мы можем создать очередь, из которой будем брать описание следующего чанка для загрузки, или просто воспользуемся счетчиком блоков:

```
// Общее количество чанков, которое надо загрузить
const chunksCount = Math.ceil(file.size / options.chunkSize);

// Количество чанков, оставшихся незагруженными
let chunksLeft = chunksCount;
```

Теперь допишем нашу функцию `uploadNextChunk`:

```
// Если мы уже прочитали весь файл, то просто завершаем работу «потока»
if (!chunksLeft) {return Promise.resolve();}

// Вычисляем смещение, с которого надо прочитать данные
const offset = (chunksCount - chunksLeft) * options.chunkSize;

// Уменьшаем количество оставшихся частей
```

```
chunksLeft--;


// И отправляем чанк на сервер
return send(
    file.read(offset, offset + options.chunkSize), offset)
        .then(uploadNextChunk())
);
```

Таким образом, после того как будет загружен текущий чанк, функция `uploadNextChunk` будет поставлена в очередь микрозадач для загрузки последующей части. В случае ошибки «поток» будет прерван и `Promise.all`, используемый в основной функции, вернет выброшенную ошибку. Но остальные потоки пока ничего не знают о произошедшей ошибке и будут отправлять чанки, пока они есть. Это не очень хорошее поведение. Чтобы этого избежать, заведем переменную `error`, содержащую признак ошибки, и при каждом последующем вызове `uploadNextChunk` будем анализировать значение. Тогда весь код будет выглядеть так:

```
export async function upload(file: Source, send: SendCb, options: Options):
Promise<void> {
  // Общее количество чанков, которое надо загрузить
  const chunksCount = Math.ceil(file.size / options.chunkSize);

  // Количество чанков, оставшихся незагруженными
  let chunksLeft = chunksCount;
  let error = false;
```

```
  await Promise.all(new Array(options.maxChunks).fill(0).map(() =>
uploadNextChunk()));


  function uploadNextChunk(): Promise<void> {
      // Если мы уже прочитали весь файл, то просто завершаем работу
«потока»
      if (!chunksLeft) {return Promise.resolve();}

      // Вычисляем смещение, с которого надо прочитать данные
      const offset = (chunksCount - chunksLeft) * options.chunkSize;

      // и уменьшаем количество оставшихся частей
      chunksLeft--;

      // И отправляем чанк на сервер
      return send(file.read(offset, offset + options.chunkSize),
offset).then(
          // если ошибка была в другом «потоке», то досрочно завершаем этот
          () => error ? Promise.resolve() : uploadNextChunk(),
          e => {
              error = true;
              return Promise.reject(e);
          }
      );
  }
```


IT's
TINKOFF

```
}
```

Вопрос 25

Напишите декоратор, который логирует переданные в метод аргументы, вернувшееся значение и время исполнения функции в заданном формате.

Пример использования:

```
class SomeService {
    @Logger
    requestSomeData(id: number, provider: string): SomeData {
        // возвращает {name: "Alice"}
        return loadData(id, provider);
    }
}

const service = new SomeService();
const result = service.requestSomeData(1, 'test provider');
```

В консоли появляется:

```
SomeService.requestSomeData: 0.033ms
  arguments: [1,"test-provider"]
  result: {"name":"Alice"}
```

РЕШЕНИЕ

Для решения этой задачи нужно вспомнить, как пишутся TypeScript-декораторы и какие три параметра в них доступны. Для замера времени выполнения можно использовать Date.now(), но можно вспомнить и о методах консоли, которые отлично подходят для подобных ситуаций: console.time и console.timeEnd.

```
function Logger(target: any, method: string, descriptor:
PropertyDescriptor) {
    // Сохраняем оригинальное значение метода в переменную savedValue
    const savedValue = descriptor.value;
    // Имя класса позволит не перепутать запуск метода с другими методами
    const className = target.constructor.name;

    // Переопределяем функцию метода, добавляя логирование времени
    // выполнения, переданных аргументов и результата.
    descriptor.value = function (...args: Array<unknown>) {
        const timeKey = `${className}.${method}`;
        console.time(timeKey);
        const result = savedValue.apply(this, args);

        console.timeEnd(timeKey);
        console.log(` arguments: ${JSON.stringify(args)}`);
        console.log(` result: ${JSON.stringify(result)}`);

        return result;
    };
}
```

Теперь при каждом вызове метода с таким декоратором в консоль будет выводиться время выполнения, переданные аргументы и результат.


IT's
TINKOFF