# Лаба 8–7: Ответы на вопросы

## Что такое callback‑функция и как она используется в JavaScript?

• Callback (обратный вызов) — это функция, передаваемая как аргумент другой функции и вызываемая позднее, когда основная операция завершается.

Применения

▪ Асинхронные операции: `fs.readFile(path, (err, data)=>{…})`

▪ Обработчики событий: `button.addEventListener('click', handler)`

▪ Итерационные методы массивов: `[1,2,3].map(x => x\*2)`

▪ Пользовательские «hooks» внутри собственных функций высокого порядка.

Преимущества — отделяют логику события от места его возникновения, позволяют не блокировать поток исполнения, образуя основу асинхронной модели JS.

## Как работает функция setTimeout и в чём её отличие от setInterval?

• `setTimeout(fn, delay)` ставит однократное выполнение fn после не менее чем delay мс; возвращает ID тайм‑аут.

• `setInterval(fn, delay)` вызывает fn периодически каждые delay мс, пока не будет остановлен.

Технические различия

1. `setInterval` помещает вызов в очередь каждую \*delay\* мс независимо от того, завершился ли предыдущий, что может привести к «накладке» функций.

2. `setTimeout` можно цепочкой вызывать рекурсивно (`function tick(){…; setTimeout(tick, delay);}`) — это упрощает динамическое изменение интервала.

3. Оба таймера реализованы браузером/движком и их коллбэки попадают в очередь событий Event Loop, поэтому фактическое время может быть больше указанного.

## Как можно отменить выполнение функции, запланированной с помощью setTimeout?

Сохраните возвращённый ID и вызовите `clearTimeout(id)`. Аналогично `clearInterval(id)` для интервала:

const id = setTimeout(alert, 5000);

clearTimeout(id);

## Почему использование callback‑функций может привести к «адскому коллбэку» (callback hell), и как этого избежать?

Callback hell — это пирамида вложенных коллбэков вида:

a(()=>{

b(()=>{

c(()=>{ … });

});

});

Проблемы: плохая читаемость, трудное тестирование, запутанная обработка ошибок.

Способы избежать

• Promises: `a().then(b).then(c).catch(err=>…)`

• async/await — синтаксический сахар над промисами, код выглядит линейно.

• Вынос коллбэков в именованные функции («не вкладывать — а ссылаться»).

• Разделять логику на модули и использовать композицию вместо вложенности.

## Каковы преимущества использования Map для хранения пар «ключ‑значение» по сравнению с обычными объектами?

• Любой тип ключа (объект, DOM‑элемент) → удобно кэшировать.

• Отсутствие коллизии с прототипом — ключ `'toString'` не перекроет метод.

• Быстрые и предсказуемые операции\*\* `get/has/set/delete` O(1).

• size даёт количество без подсчёта.

• Лёгкая конвертация\*\* к массиву `Array.from(map)` → методы `map/filter`.

• Итерабельность по умолчанию (`for…of` даёт `[key,value]`).

## Какие методы предоставляет класс Set, и как они могут быть полезны для работы с уникальными значениями?

• `add(value)` — добавление (возвращает сам Set для чейнинга).

• `has(value)` — проверка существования.

• `delete(value)` / `clear()` — удаление одного или всех.

• `size` — количество элементов.

• `values()` `keys()` `entries()` — итераторы.

• `forEach(cb)` — перебор.

Примеры: дедупликация массива (`new Set(arr)`), отслеживание уникальных посетителей, быстрый алгоритм поиска пересечений (`intersection = new Set([...a].filter(x=>b.has(x)));`).

## Как использовать методы forEach, map и filter с коллекциями?

• `Map` и `Set` имеют собственный `forEach(cb, thisArg)`:

mySet.forEach(v => console.log(v));

myMap.forEach((val, key) => { … });

• `map` и `filter` \*\*не\*\* встроены, но легко применимы после преобразования в массив:

const doubled = [...mySet].map(x=>x\*2);

const adults = new Map([...users].filter(([id,u]) => u.age>=18));

• Вместо `filter` для Set часто используют прямой перебор + `delete` для удаления лишних элементов.”