JS. ЧИСЛА

В JS число = 64 бита (формат IEEE-754):
- 52 для хранения цифр
- 11 для точки
- 1 для знака (+/-)

Число слишком большое = переполнит 64-битное хранилище = Infinity

Почему 0.1 + 0&2 !== 0.3:

Числа хранятся в памяти в бинарной форме. Что такое 0.1? 1/10. В десятичной СС 1/10 легко представить, а 1/3 становится бесконечной дробью. Аналогично в двоичной СС деление на 2 работает, а 1/10 становится бесконечной дробью.
В JS нет возможности хранить 0.1 и 0.2 используя двоичную СС, поэтому формат IEEE-754 округляет до ближайшего возможного числа. Округление не позволяет нам увидеть неточности, но они существуют. И когда мы суммируем 2 числа, их неточности тоже суммируются.

Решение:

- строковое представление (.toFixed(кол-во цифр))
- хранение суммы в центах

03. НАБЛЮДАТЕЛЬ

— объект, информирующий наблюдателей об изменении состояния.

Наблюдатель	Коллбэки
Несколько наблюдателей	Один наблюдатель (т.е. коллбэк)

ООП	Node.js
• интерфейсы • конкретные классы • иерархию	Уже встроен и доступен через класс EventEmitter.

EventEmitter:
- добавление функций-наблюдателей
- их вызов при срабатывании события
- часть модуля events

См. 02-15-event-emitter.js

Методы EventEmitter	
• on (once/addListener/prependListener/ prependOnceListener) — не делает проверки на повтор ФН • once — удаляет ф.-наблюдатель и выполняет ее • off (removeListener/removeAllListeners)	• вернут экземпляр EventEmitter • this внутри функци (не стрелочной) ссылается на экземпляр EventEmitter
emit	вернет true, если были обработчики на событие, иначе false
EventEmitter.defaultMaxListeners (+ process.on('warning'))	переписать развешенное кол-во ФН для всех экземпляров EE
listeners()	статический массив ФН
removeAllListeners([eventName])	плохая практика удалять все ФН
removeListener(eventName, listener)	удаляет самую последнюю ФН удаление ФН внутри ФН не удаляет их из текущего исполнения

- функции-наблюдатели вызываются синхронно — для асинхронного исполнения используется process.nextTick() или setImmediate()
- данные, возвращаемые return, игнорируются
- всегда добавлять обработчик на событие error
- передача ошибки осуществляется при помощи события error. Без ФН такого события ошибка будет проброшена в Цикл Событий
- *при добавлении новой функции-наблюдателя вызывается внутреннее событие newListener, наблюдателям которого предаются имя события и ссылка на ФН

- Если необходимо предоставить доп. функционал, создается новый класс на базе EventEmitter.

См. 02-16-event-emitter.js — 02-17-event-emitter.js

Вызов событий	
Синхронно	Асинхронно
Необходимо назначать ФН до начала события	Есть время назначить ФН

См. 02-18-event-emitter.js

Использование	
Коллбэки	EventEmitter
асинхронно	обработать то, что только что произошло
• готовность • количество кода • поддержка разных событий • семантика (частота события) • вызов одной или нескольких функций	
Совместное использование: https://github.com/isaacs/node-glob/blob/8fa8d561e08c9eed1d286c6a35be2cd8123b2fb7/glob.js	

Дополнительная литература:
- https://nodejs.org/api/events.html#events_emitter_emit_eventname_args

03. АСИНХРОННЫЙ ПОРЯДОК ИСПОЛНЕНИЯ НА КОЛЛБЭКАХ

Замыкания и анонимные функции полностью соответствуют принципу KISS. Однако пренебрежение модульностью и компонентами, которые легко переиспользовать и поддерживать приведет к разрастанию функции и плохой организации кода.

См. 03-01-web-spider

Callback Hell (Pyramid of Doom) — самый распространенный анти-паттерн:
- чрезмерное присутствие коллбэков
- плохая читаемость
- повторение имен в замыкании (например, err)
- замыкания могут создать утечки памяти, которые нелегко отследить

Правила:
- не злоупотреблять замыканиями
- выходить из кода как можно раньше вместо длинного if … else
- давать имена коллбэкам, хранить их вне замыкания и передавать им результат выполнения в качестве аргумента
- разделять код на небольшие переиспользуемые компоненты (модульность)

См. 03-02-web-spider

03.01 Последовательное исполнение
- последовательное исполнение операций без придачи результата:
  - следующая операция вызывается после выполнения асинхронной операции
  - использует модульность
    См. 03-02-web-spider/spider-fixed.js
- использование результата предыдущей операции как входных параметров для следующей
- перебор коллекции выполняя асинхронную операцию на каждом ее элементе

См. 03-03-web-spider

Шаблон:

```
function iterate(index) {
  if(index === tasks.length) {
    return finish();
  }
  const task = tasks[index];
  task(() => {
    iterate(index + 1);
  });
}

function finish() {
  // Вызовется, когда все задания будут выполнены.
}

iterate(0);
```

- перебор массива

- передача результата текущей операции в следующую
- выход из цикла при выполнении условия

03.02 Параллельное исполнение

— исполнение набора операций, порядок которых не важен, а важен завершение.

Параллельно в Node.js ≠ одновременно = неблокирующее исполнение

См. 03-04-async-flow.png

Поэтому синхронные операции должны чередоваться с асинхронными для недопущения блокируемости.

См. 03-05-web-spider

Шаблон:

```
const tasks = [];
  let completed = 0;
  tasks.forEach((task) => {
    task(() => {
      if (++completed === tasks.length) {
        finish();
      });
});

function finish() {
  // Вызовется, когда все задания будут выполнены.
}
```

Состояние гонки:

— здесь, параллельное исполнение нескольких задач, приводящее к нежелательным последствиям.

В других языках существуют специальные механизмы мониторинга состояний гонки, в Node.js они могут происходить часто.

См. 03-05-web-spider

Решение:

Единое хранилище информации о выполнении функции.

03.03 Ограниченное параллельное исполнение:

DoS (Denial of Service) атаки — перегрузка приложения из-за большого количества выполняемых задач. Поэтому, необходимо ограничивать кол-во операций, которые можно выполнять одновременно.

```
const tasks = [];

let concurrency = 2, running = 0, completed = 0, index = 0;

function next() {
  while (running < concurrency && index < tasks.length) {
    const task = tasks[index++];

    task(() => {
      if (completed === tasks.length) {
        return finish();
      }

      completed++;
      running--;

      next();
    });

    running++;
  }
} next();
function finish() {
  // Вызовется, когда все задания будут выполнены.
}
```

См. 03-06-task-limiting — 03-07-web-spider (можно применить на загрузке ссылок со странице, но количество загружаемых ссылок будет расти с каждым запросом. Поэтому необходимо глобальное отслеживание загрузок. На помощь может прийти **очередь**:

- динамически добавлять новые задачи
- единое место регулирования ограничений на кол-во выполняемых задач

).