گروه چهارم سوالات جاوااسکریپت

Functions

توی این تمرین میخوایم یه سری توابع خاصی که بعدا استفاده میشن رو بنویسیم و کمی با توابع و pattern هایی که میتونیم استفاده کنیم، کار کنیم.

اول لینک‌های زیر رو بخونید، این لینک‌ها، فرض میکنن شما بلدید یک تابع تعریف کنید، بهش ورودی بدید و صداش بزنید و خروجی خودتون رو دریافت کنید. پس لینکایی که اینجا هست، کمی پیشرفته تر و عمیق تره.

https://javascript.info/function-expressions#function-is-a-value
https://javascript.info/javascript-specials
https://javascript.info/currying-partials
https://javascript.info/settimeout-setinterval
https://javascript.info/function-object
https://javascript.info/recursion

بعد از خوندن این لینک‌ها، باید بدونیم تابع چیه، چطوری ورودی بدیم و خروجی بگیریم، چطوری تابع رو صدا بزنیم، چطوری توابع رو باهم دیگه ترکیب کنیم، چطوری خود توابع (و نه خروجیشون) رو به عنوان متغیر یا خروجی یه تابع دیگه پاس کاری کنیم، توابع خودشون نوعی آبجکت هستن و میتونیم بهشون پراپرتی بچسبونیم ، توابع بازگشتی چین و چطوری میشه نوشتشون و همینطور از setTimeout و setInterval استفاده کنیم.

بریم سراغ تمرین ها:

1 - تابع calculate رو بنویسید به طوری که کدهای زیر (دقیقا به همین شکل و نه حالت دیگه ای) به درستی اجرا بشن.

```
calculate(2)(3)('+') // 5
calculate(2)(3)('-') // -1
calculate(2)(3)('*') // 6
```

```
calculate(2)(3)('/') // 0.66
```

پیاده سازی چهار عمل اصلی (مشابه مثال ها) کافیه.
عددهای 2 و 3، صرفا جنبه مثال دارن و تابع باید با عددهای مختلف به درستی کار بکنه.

2 - تابع sumFromTo رو به صورت **بازگشتی** بنویسید. این تابع، 2 تا عدد به عنوان ورودی میگیره و جمع عددها از ورودی اول تا ورودی دوم رو حساب میکنه. مثلا:

```
sumFromTo(1, 100) // 5050
sumFromTo(1, 5) // 11
sumFromTo(-5, 5) // 0
```

3 - یه تابع میخوایم به اسم callOnceAfter بنویسیم. این تابع به عنوان ورودی، یه تابع دیگه و یک زمان (بر حسب ثانیه) رو دریافت میکنه. به عنوان خروجی هم بهمون یه تابع جدید میده.
ویژگی این تابع خروجی اینه که اگه پشت سر هم صداش کنیم، فقط بعد از آخرین باری که صداش کردیم و بعد از گذشت اون مقدار زمان، تابعی که بهش ورودی دادیم رو صدا میزنه. مثلا:

```
Function logName() { console.log("ali") }
const lazyLogName = callOnceAfter(logName, 2)
lazyLogName()
lazyLogName()
lazyLogName()
lazyLogName()
lazyLogName()
lazyLogName()
lazyLogName() // last call

// logs "ali" after 2 seconds after last call
```

از ایده این سوال، بعدا برای حذف تغییرات زائد هنگام تایپ کردن کلمات توی اینپوت ها استفاده میکنیم که بهش میگیم debounce (ولی سرچش نکنید چون جوابش هست تو اینترنت)

4 - یه تابع میخوایم بنویسیم به اسم limitCallForEveryDuration، این تابع، تا حدودی شبیه تابع سوال قبل (callOnceAfter) هست. این تابع هم 2 تا ورودی میگیره که اولی یه تابع دیگه هست و دومی یه عدد به ثانیه هست. این تابع، عنوان خروجی، یه تابع دیگه مشابه سوال قبل بهمون برمیگردونه.
این تابع خروجی، ویژگی اش اینه که اگه پشت سر هم صداش کنیم، توی هر x ثانیه (ورودی دوم)، فقط یه بار اجرا میشه. به بیان دقیق‌تر، از اگه از آخرین زمانی که اجرا شده، x ثانیه گذشته باشه، دوباره تابع اصلی مارو صدا میکنه. مثلا:

```
Function logName() { console.log(“ali”) }
const limitedLogName = limitCallForEveryDuration(logName, 0.5)

for(let i = 0; i < 10000; i++) {
      limitedLogName()
}
```

خروجی کد بالا: باید یه تعدادی لاگ "ali" توی کنسول ببینید. خروجی بر اساس سرعت سیستمتون متفاوته. مثلا اگه کل حلقه 2 ثانیه طول بکشه، شما 4 یا 5 بار لاگ "ali" رو می بینید. (**حدودا** به تعداد 2 ثانیه تقسیم بر 0.5 ثانیه)

از ایده این سوال، بعدا برای جلوگیری از کندی برنامه و عدم اجرای کدهایی که نسبت به قبل، تغییر قابل توجهی نداشتن، استفاده میکنیم (مثلا برای تشخیص بزرگ/کوچک شدن عرض صفحه) که به این کار میگیم throttle (ولی سرچش نکنید جوابش هست تو اینترنت).

5 - میخوایم یه تابع بنویسیم به اسم rememberOrDo که به عنوان ورودی به تابع دیگه رو میگیره و به عنوان خروجی هم بهمون یه تابع جدید برمیگردونه. ویژگی این تابع خروجی، اینه که نتایج صدا زدن های قبلیش رو نگه میداره و اگه قبلا با یه سری ورودی صدا زده شده بود، خروجی ای که از قبل ذخیره کرده بوده رو بهمون نشون میده و دوباره کاراشو انجام نمیده. مثلا:

```
Function slowSum(a, b) { return a + b }
Const fastSum = rememberOrDo(slowSum)
```

اینجا 2 و 3 براش جدیده، پس جواب رو واقعا حساب میکنه، جواب رو ذخیره میکنه و // `fastSum(2, 3)` جواب رو برمیگردونه

اینجا هم 5 و 6 براش جدیده، پس جواب رو واقعا حساب میکنه، جواب رو ذخیره میکنه // fastSum(5, 6)
.و جواب رو برمیگردونه
اینجا، چون قبلا 2 و 3 رو دیده، دیگه دوباره جمعشون نمیکنه و میره ببینه قبلا جوابش // fastSum(2, 3)
چی بوده، همونو برمیگردونه
.عین قبلی. از این به بعد تا ابد، 2 و 3 رو جوابش رو میدونه و دوباره جمعشون نمیزنه // fastSum(2, 3)

فرض کنید ورودی هایی که به fastSum (خروجی تابع سوال) میدیم، همیشه رشته یا عدد هستن.
حواسمون باشه که هر تعداد ورودی که slowSum میگرفت، باید به fastSum هم بتونیم بدیم.

از ایده این سوال، بعدا مفاهیمی مثل cache (در نرم افزار) و memo (در ری اکت) بوجود میاد که بسیار کاربردی و مهم هستن. به این معنا که اگر کار پرهزینه ای رو قبلا انجام دادیم، دیگه انجامش ندیم و از نتیجه قبلی استفاده کنیم.

6 - یه تابع بنویسید به اسم makeCancelableTimeout. این تابع، ورودی هاش دقیقا مشابه setTimeout هست. ینی یه تابع ورودی میگیره و یه عدد (معنی زمان بر حسب میلی ثانیه). این تابع، علاوه بر اینکه اون timeout مون رو برامون ست میکنه، به عنوان خروجی، یه تابع بهمون میده که اگه صداش کنیم، تایم اوت رو کنسل میکنه.

```
Function log() { console.log(“hadi”) }
Const cancel = makeCancelableTimeout(log, 2000)
cancel()
```

.بعد از اجرای این کد، دیگه نباید چیزی لاگ بشه و تایم اوتی که ست میکنیم عملا باید کنسل بشه

از ایده این سوال، بعدا برای کنسل کردن کارهایی که خیلی طول کشیدن (درخواست http یا event ها) استفاده میشه.

برای تمرین بیشتر:

- یه تابع به صورت بازگشتی بنویسید که فاکتوریل عدد ورودی رو حساب کنه.
- یه تابع بنویسید که یه 2 تا عدد بگیره و یه زمان (x)، و هر x ثانیه، به ترتیب عددها رو از شروع تا پایان (2 عدد ورودی) پرینت بکنه.
- تابع sum رو جوری بنویسید که با هر تعداد پرانتز کار کنه و بینهایت بشه صداش کرد:

- `sum(1)(2) == 3; // 1 + 2`
- `sum(1)(2)(3) == 6; // 1 + 2 + 3`
- `sum(5)(-1)(2) == 6`
- `sum(6)(-1)(-2)(-3) // 0`
- `sum(0)(1)(2)(3)(4)(5) // 15`