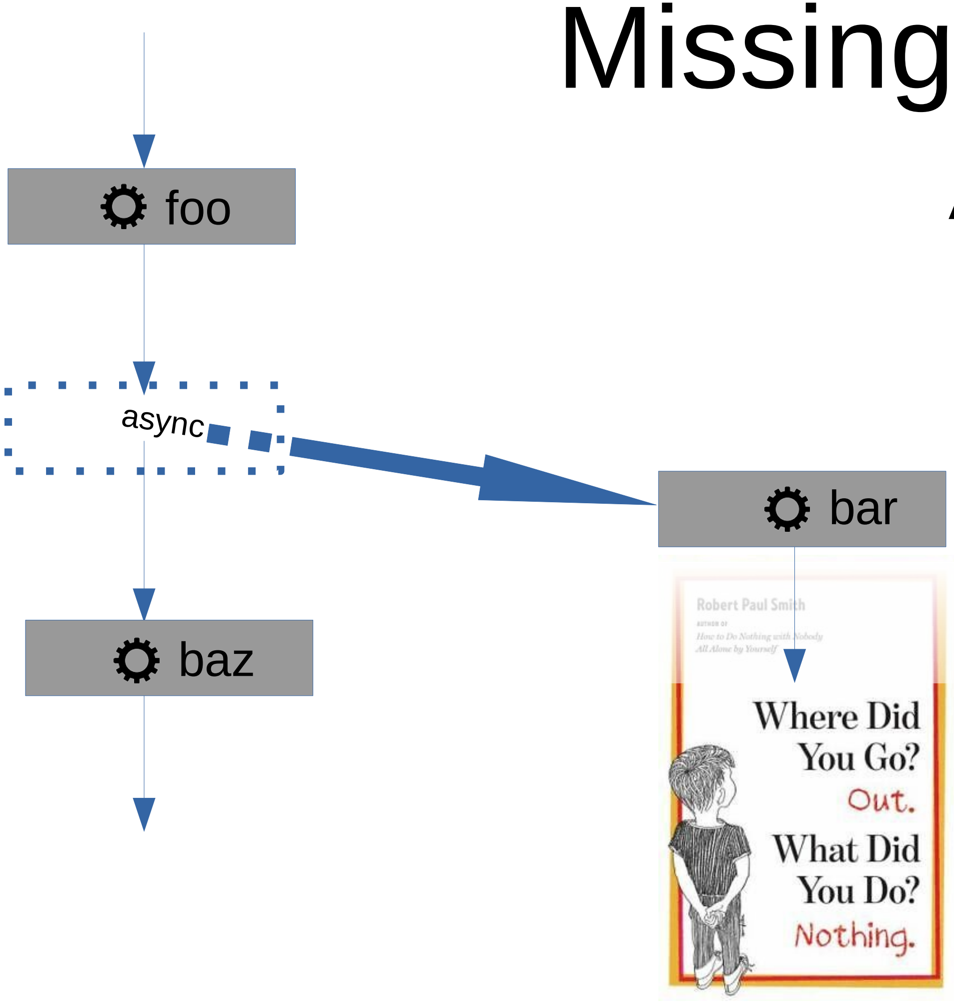
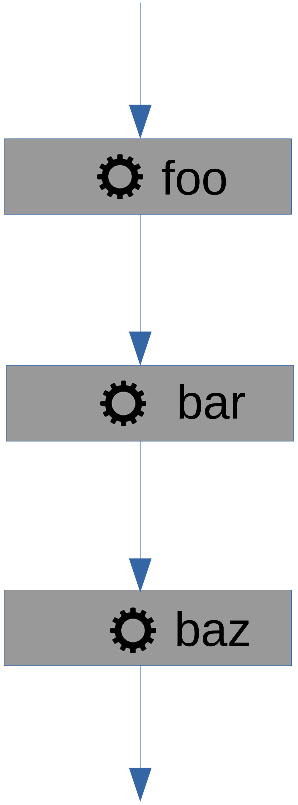
# Missing Await

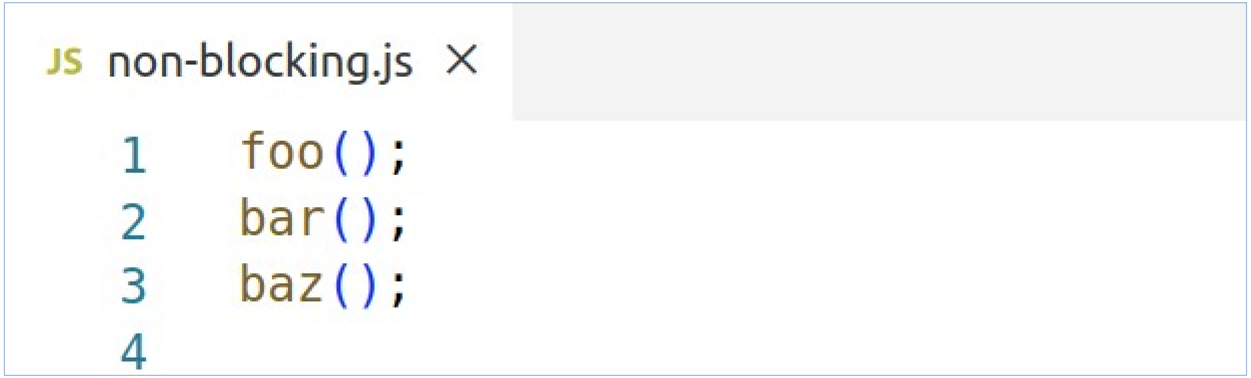
*( Promises, concurrency, synchronization, and a classic bug )*



KWJS talk

# JavaScript is a single non-blocking execution thread

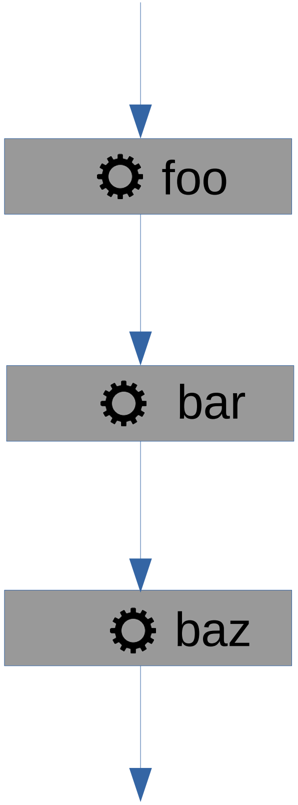


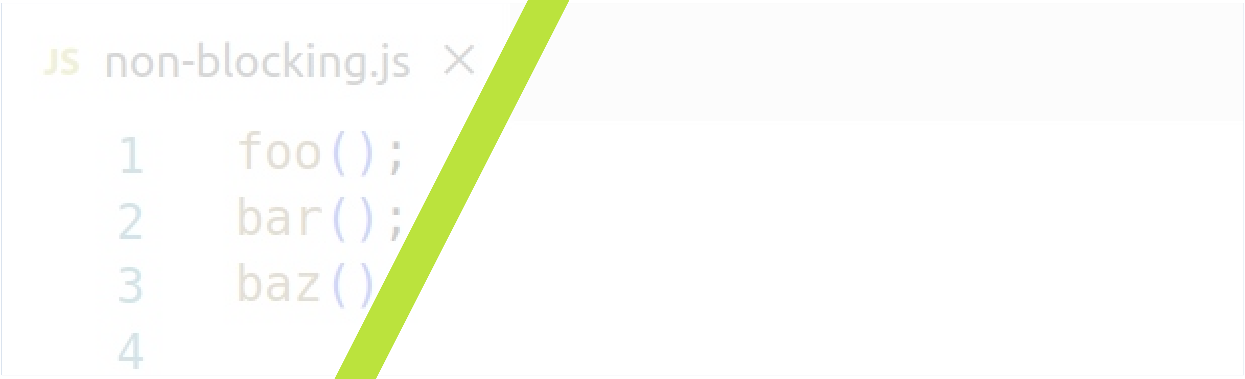


```
foo();
bar();
baz();

```

# JavaScript is a single non-blocking execution thread

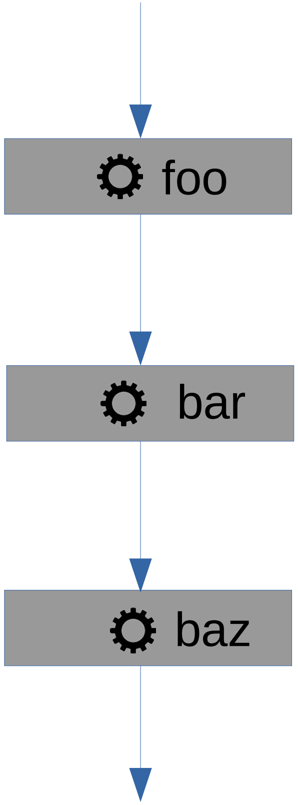


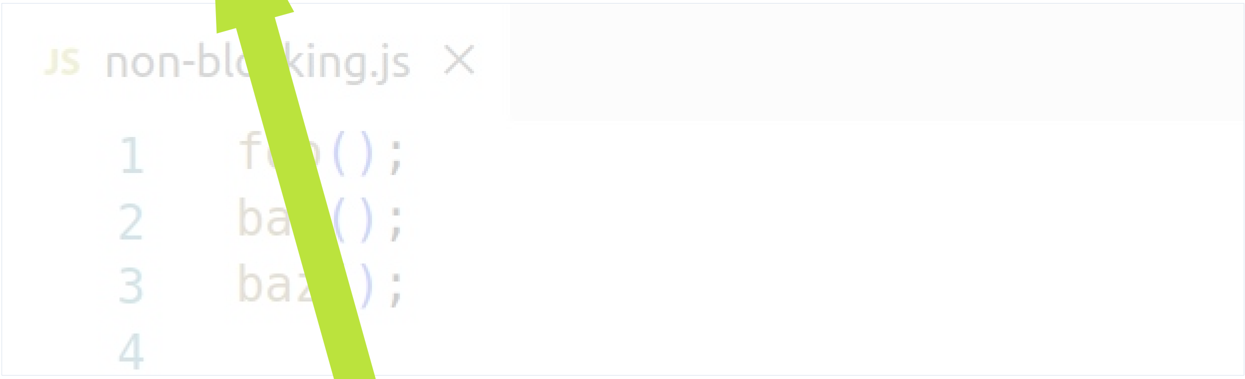


```
foo();
bar();
baz()
```

Kinda restrictive, cause only simple execution flow is possible

# JavaScript is a single non-blocking execution thread

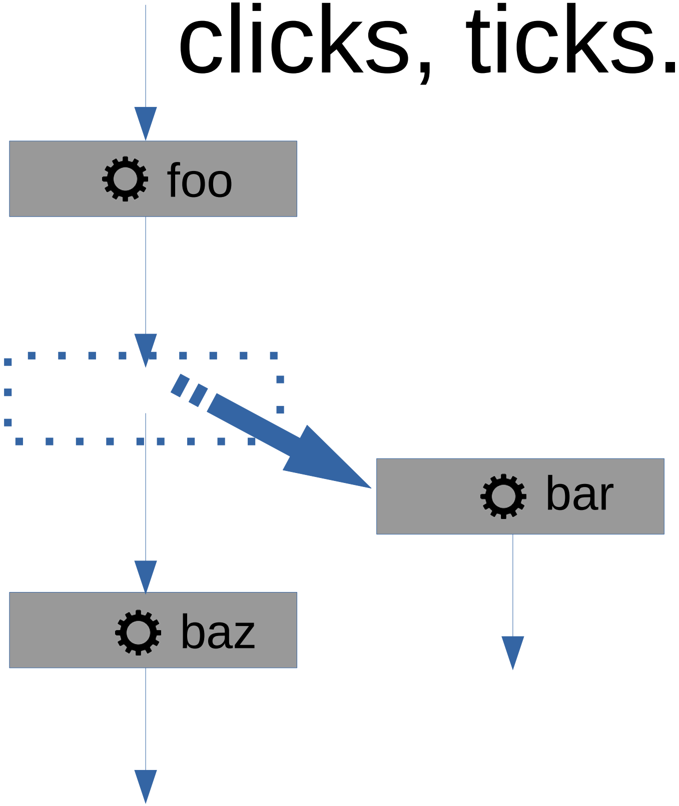


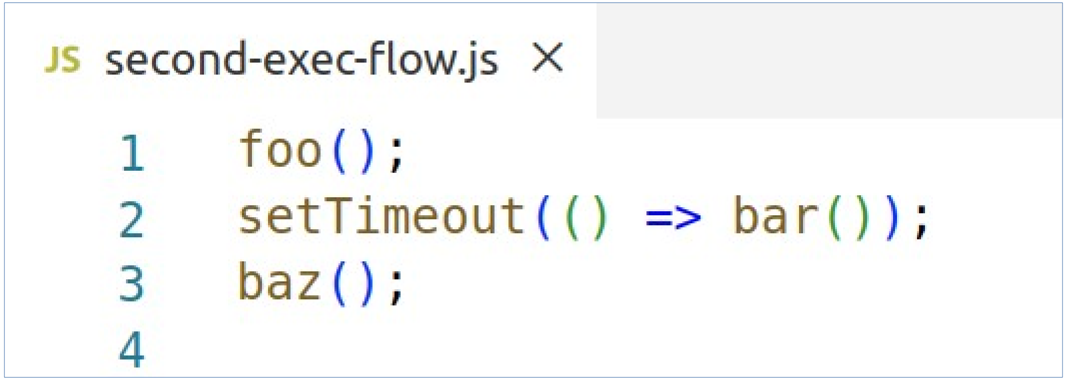


```
foo();
bar();
baz();
```

Can’t wait sanely for external components (server, disk, etc.)
Can only do tricks with callbacks

# JavaScript execution starts from events like clicks, ticks. Hence, setTimeout() trick

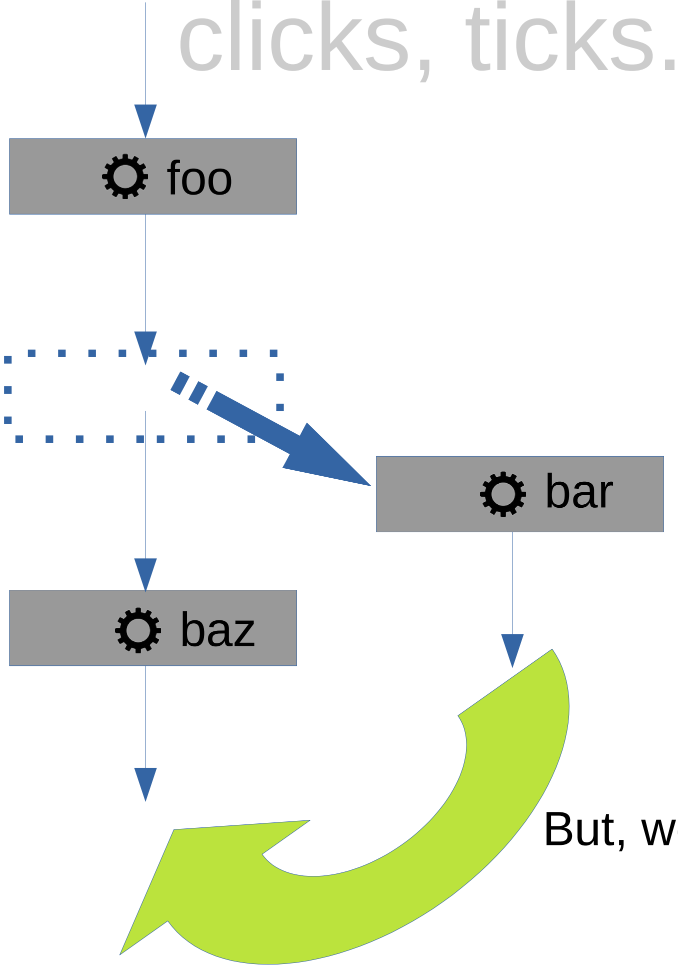


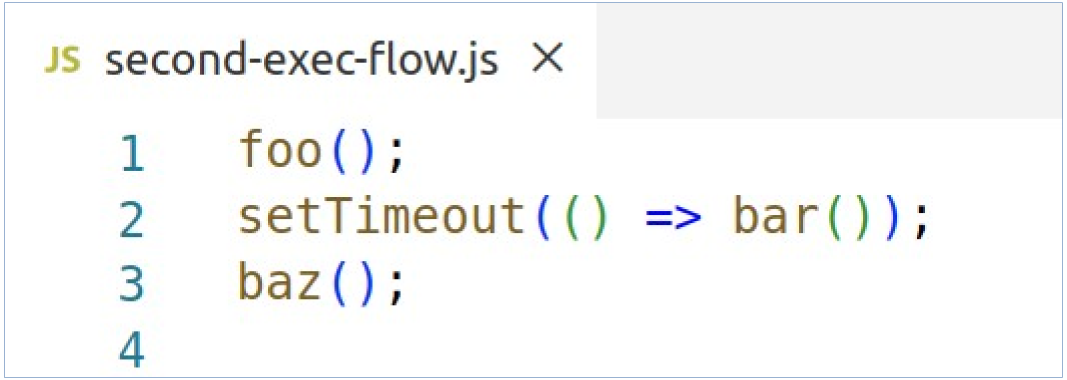


```
foo();
setTimeout(() => bar());
baz();

```

# JavaScript execution starts from events like clicks, ticks. Hence, setTimeout() trick



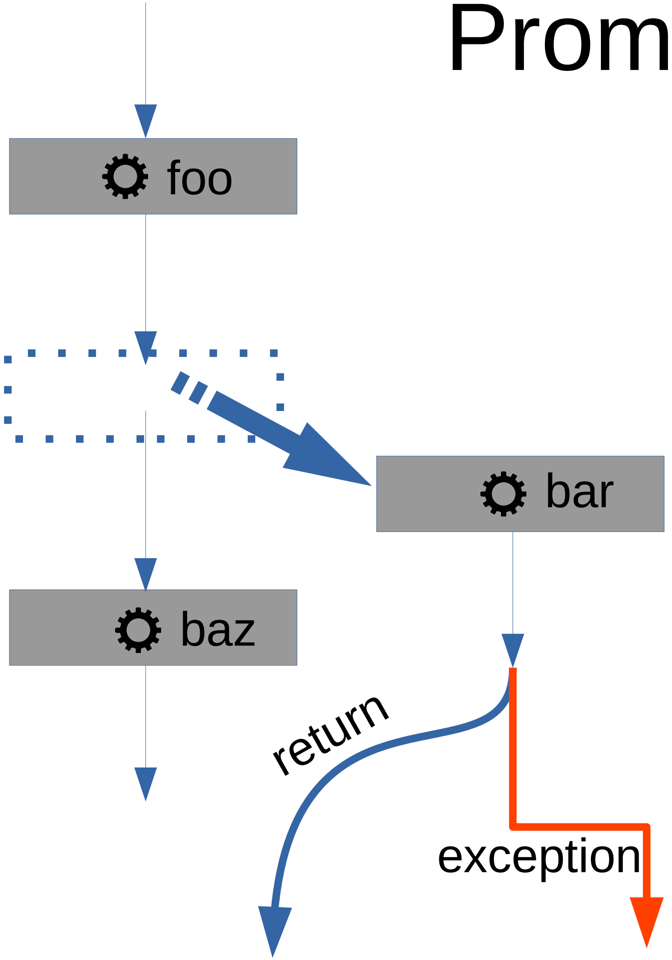


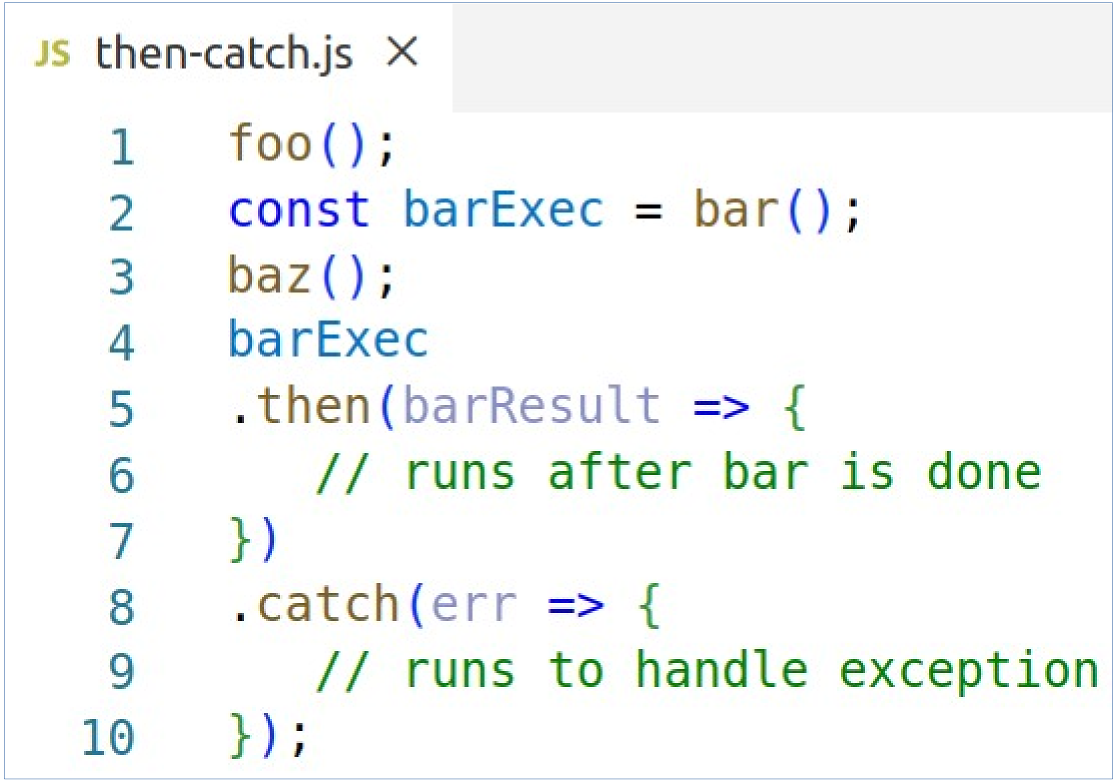
```
foo();
setTimeout(() => bar());
baz();

```

But, we can’t join these flows ... sanely

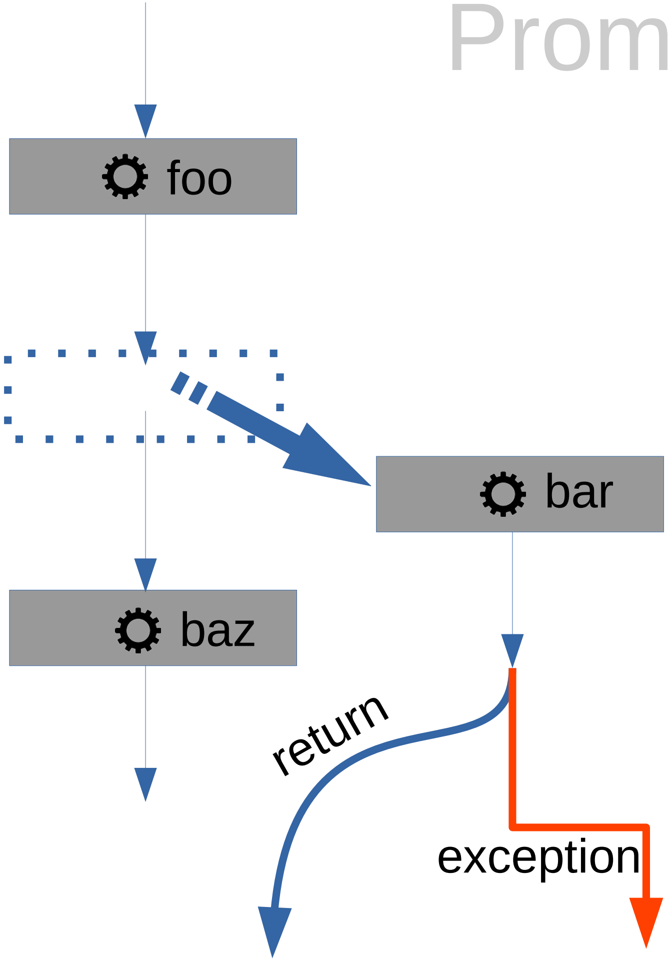
# Second order object with execution tail: Promise with then() and catch()

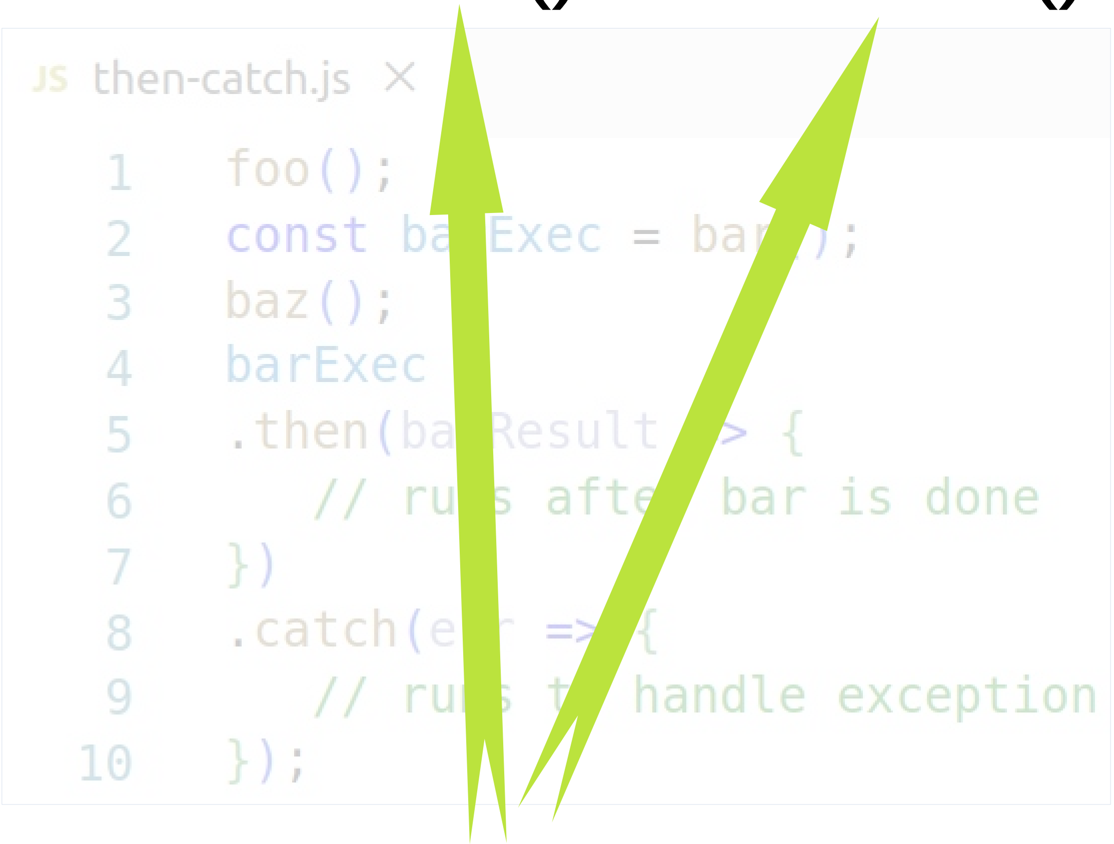




```javascript
foo();
const barExec = bar();
baz();
barExec
.then(barResult => {
    // runs after bar is done
})
.catch(err => {
    // runs to handle exception
});
```

# Second order object with execution tail: Promise with then() and catch()

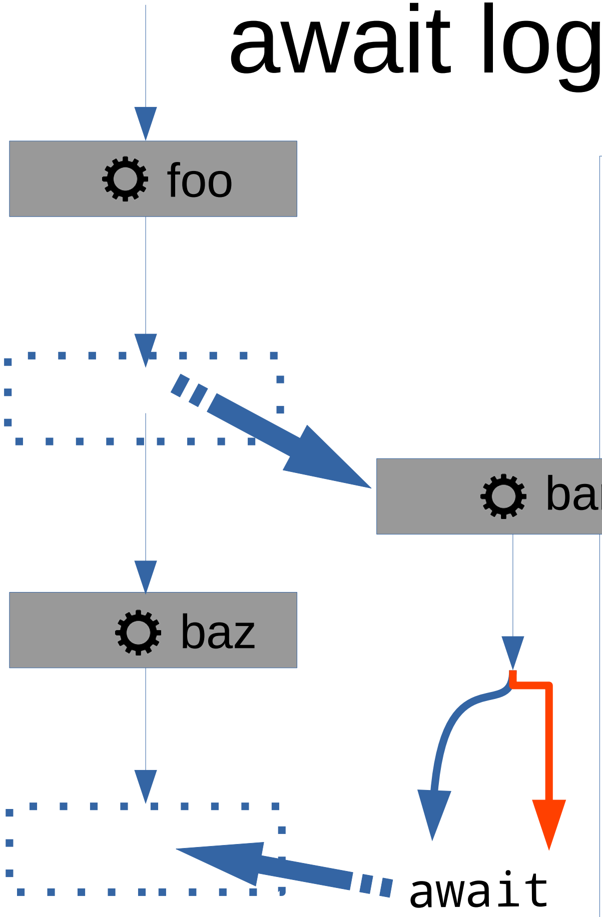


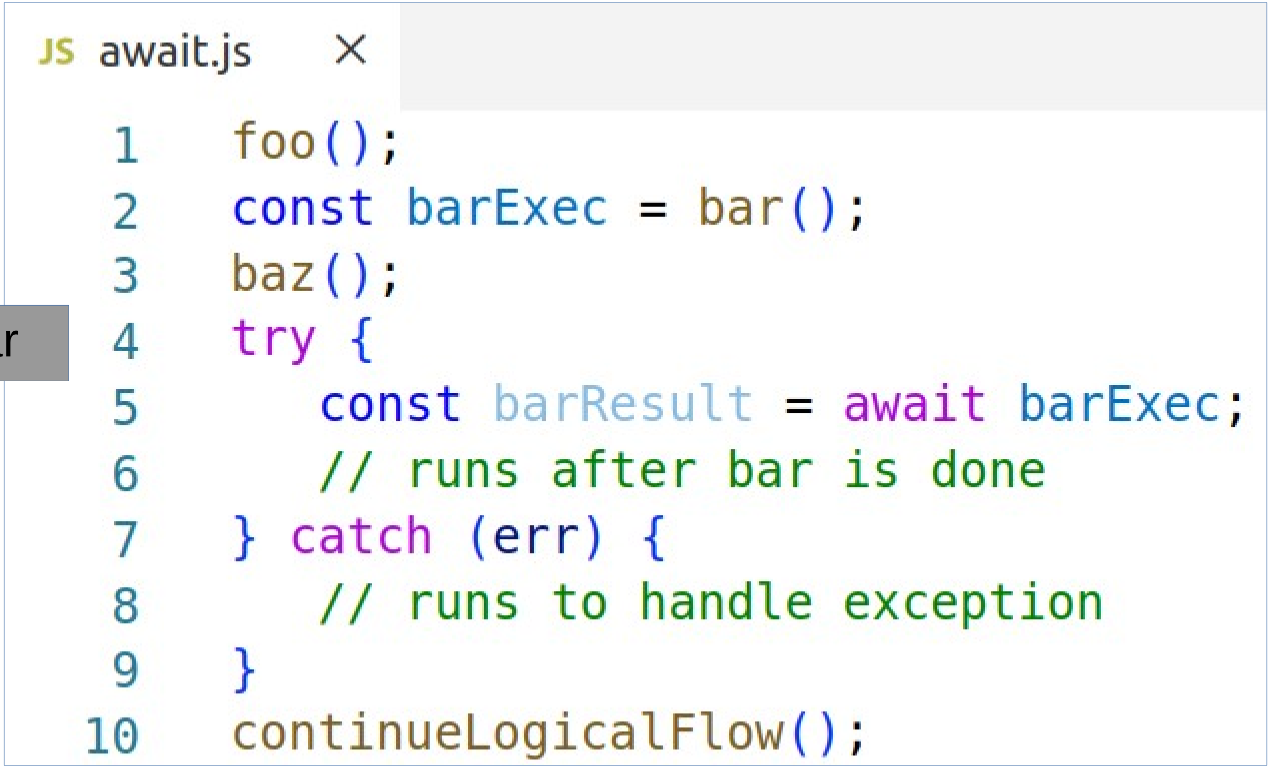


```
foo();
const barExec = bar();
baz();
barExec
.then(barResult => {
    // runs after bar is done
})
.catch(err => {
    // runs to handle exception
});
```

Execution events? Hence, callbacks, callbacks

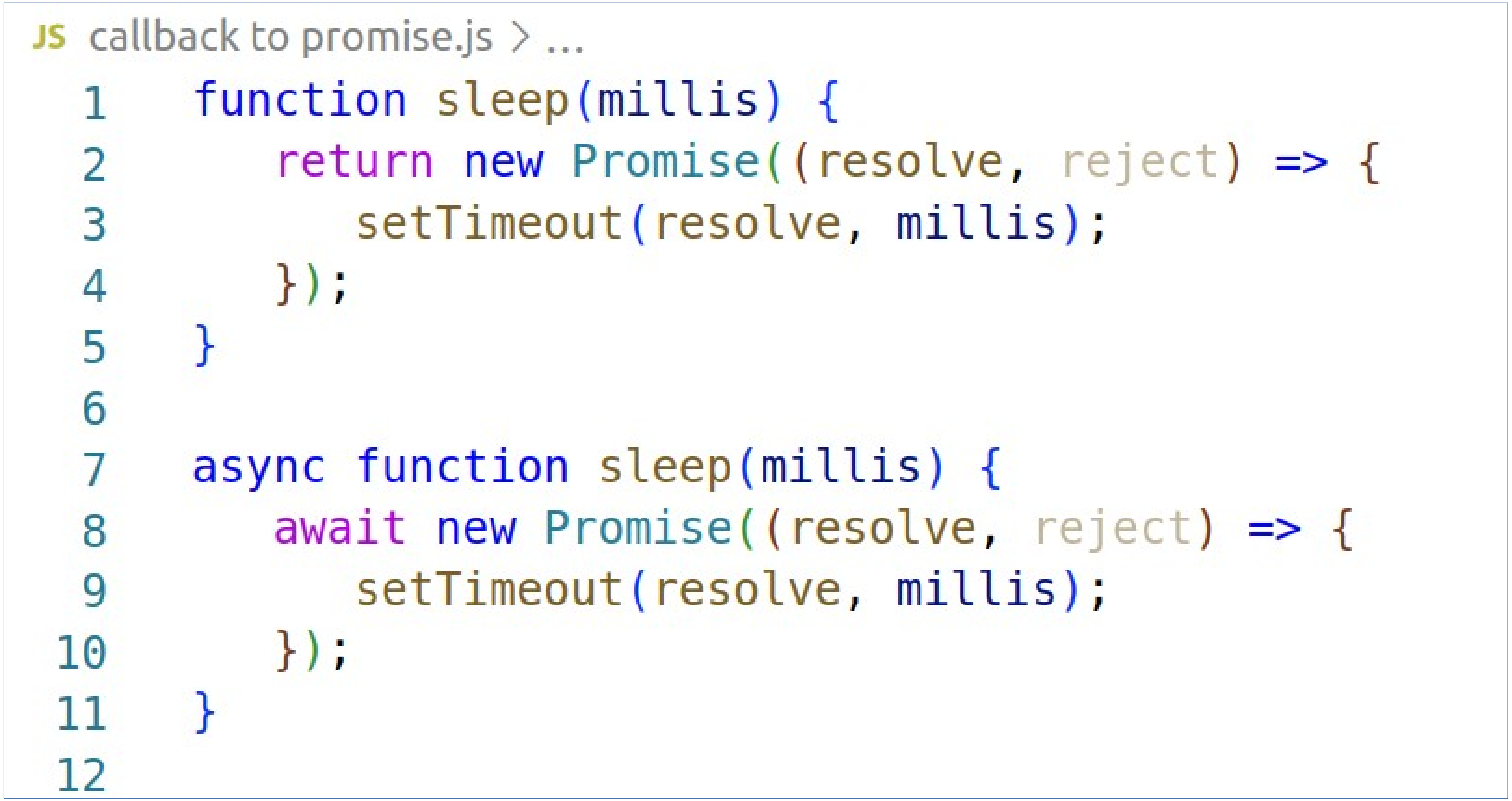
# Tie execution tail into logical thread: await logically, don’t block vm thread





```
foo();
const barExec = bar();
baz();
try {
    const barResult = await barExec;
    // runs after bar is done
} catch (err) {
    // runs to handle exception
}
continueLogicalFlow();
```

# Found useful: turning callbacks’ process flow into more explicit promise flow

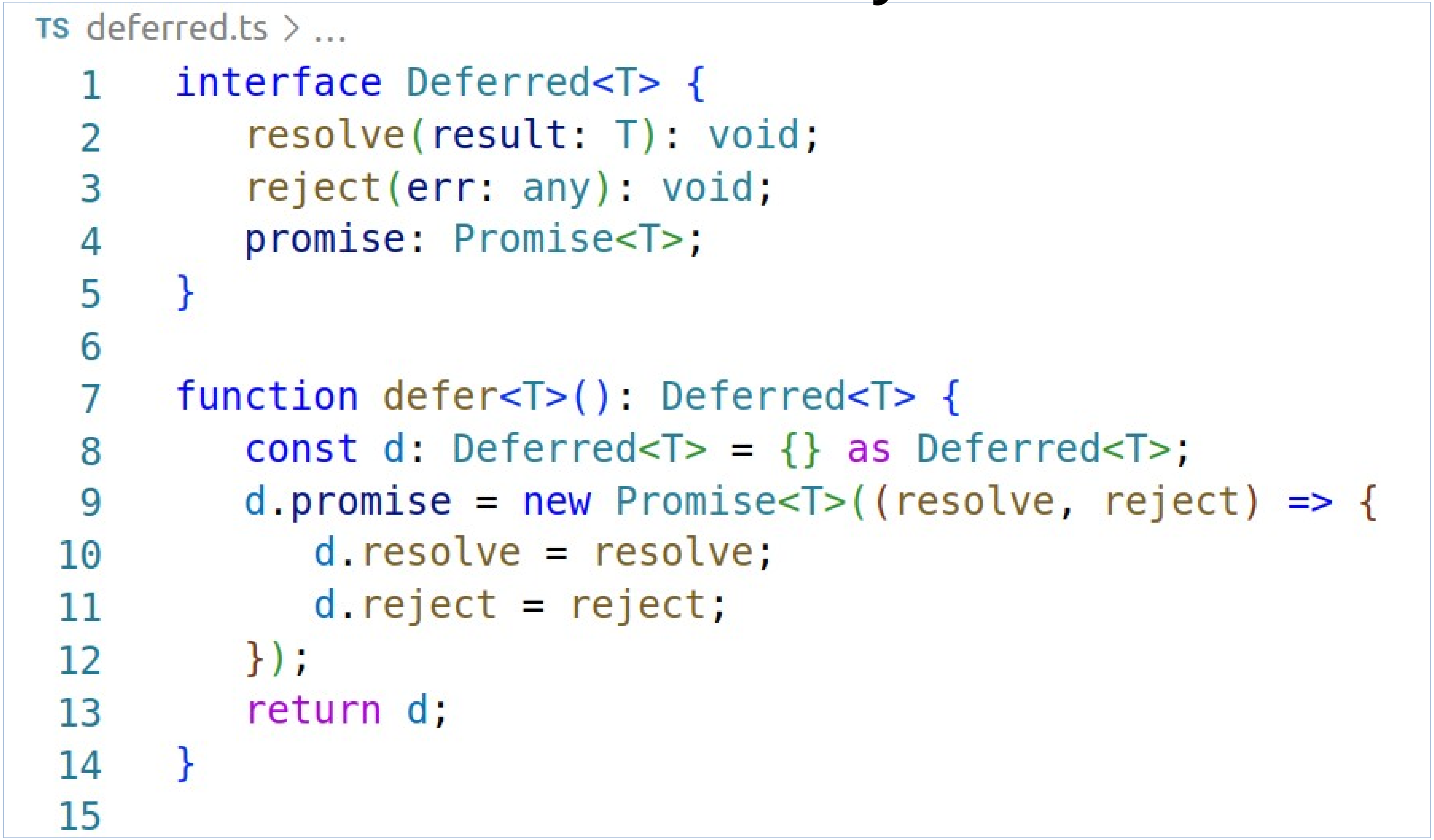


```
function sleep(millis) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, millis);
    });
}

async function sleep(millis) {
    await new Promise((resolve, reject) => {
        setTimeout(resolve, millis);
    });
}

```

# Found useful: sometimes you need deferred

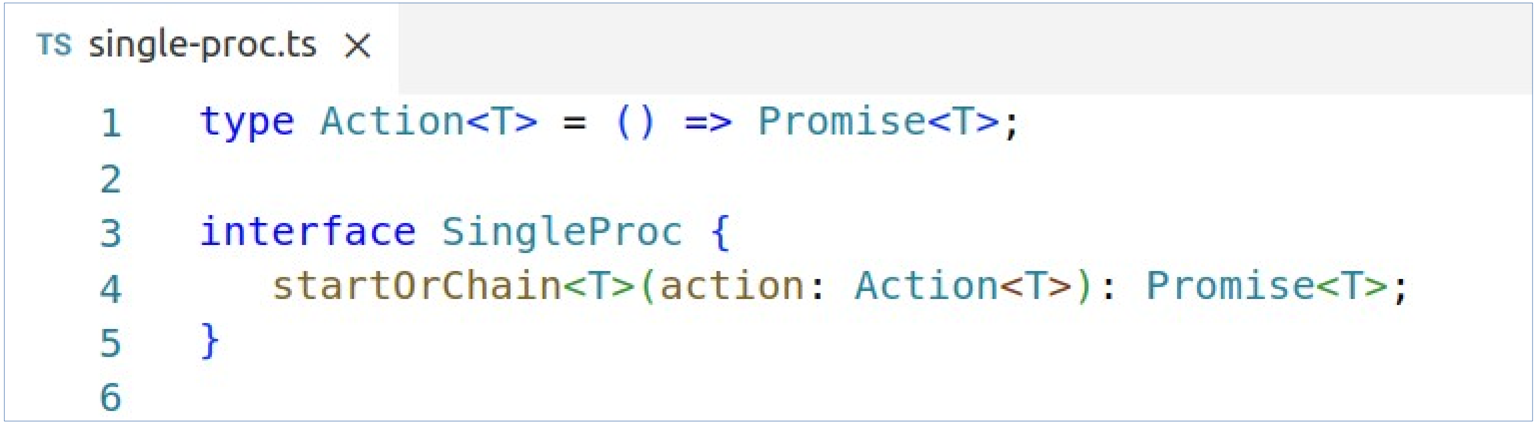


```
interface Deferred<T> {
   resolve(result: T): void;
   reject(err: any): void;
   promise: Promise<T>;
}

function defer<T>(): Deferred<T> {
   const d: Deferred<T> = {} as Deferred<T>;
   d.promise = new Promise<T>((resolve, reject) => {
      d.resolve = resolve;
      d.reject = reject;
   });
   return d;
}

```

# Found useful: synchronization of actions by ordering on a single process chain

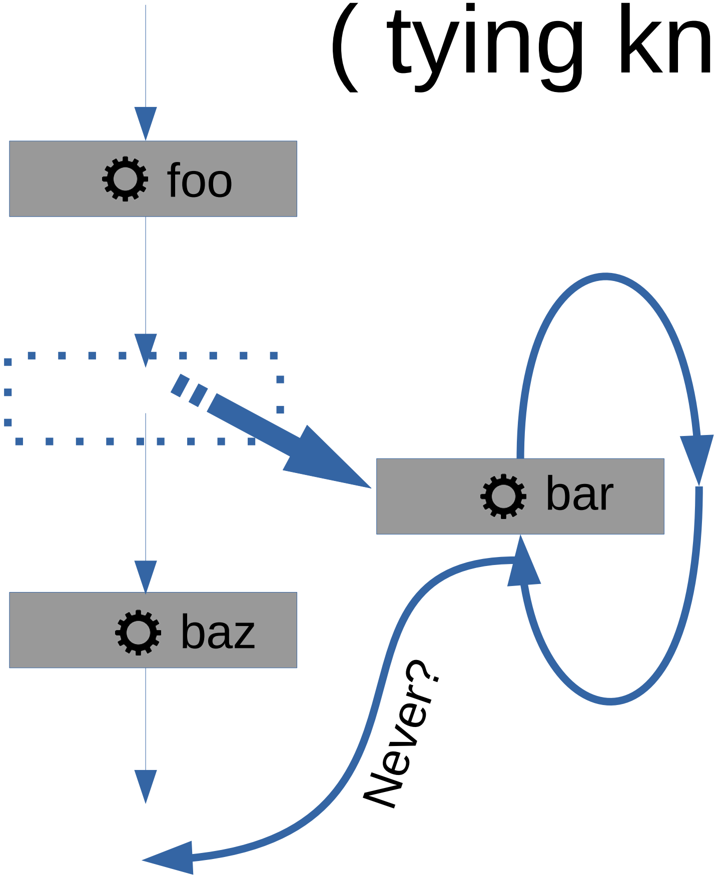


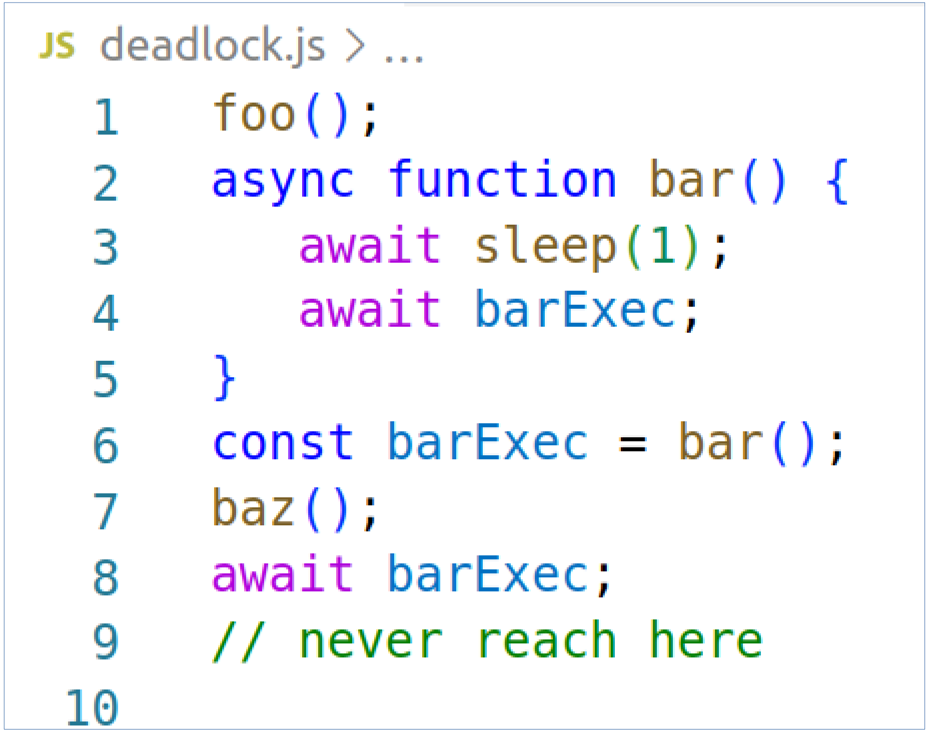
```
type Action<T> = () => Promise<T>;

interface SingleProc {
   startOrChain<T>(action: Action<T>): Promise<T>;
}

```

# Can even deadlock a non-blocking JS
# ( tying knots with logical ordering )

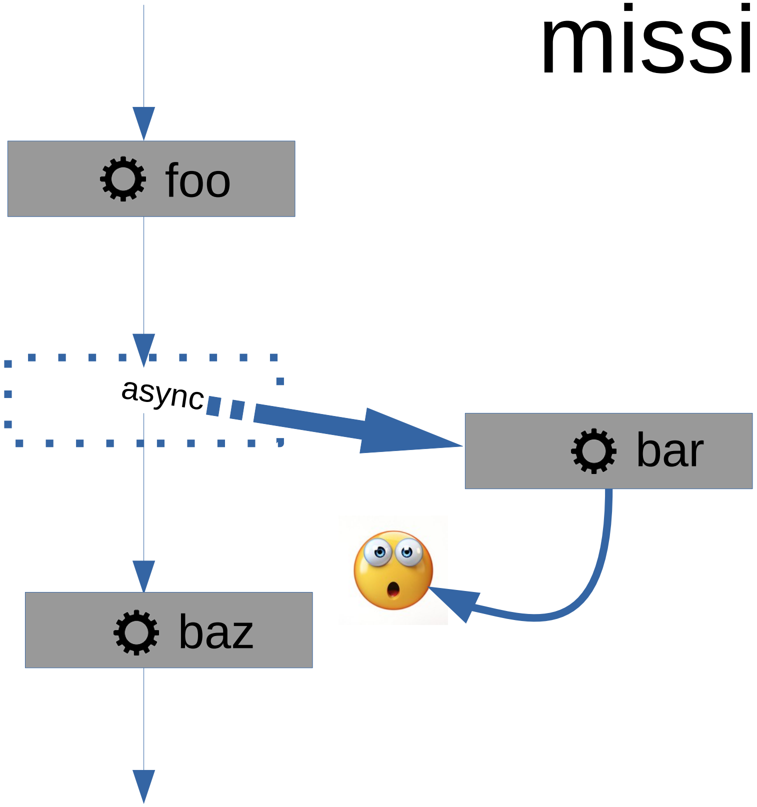


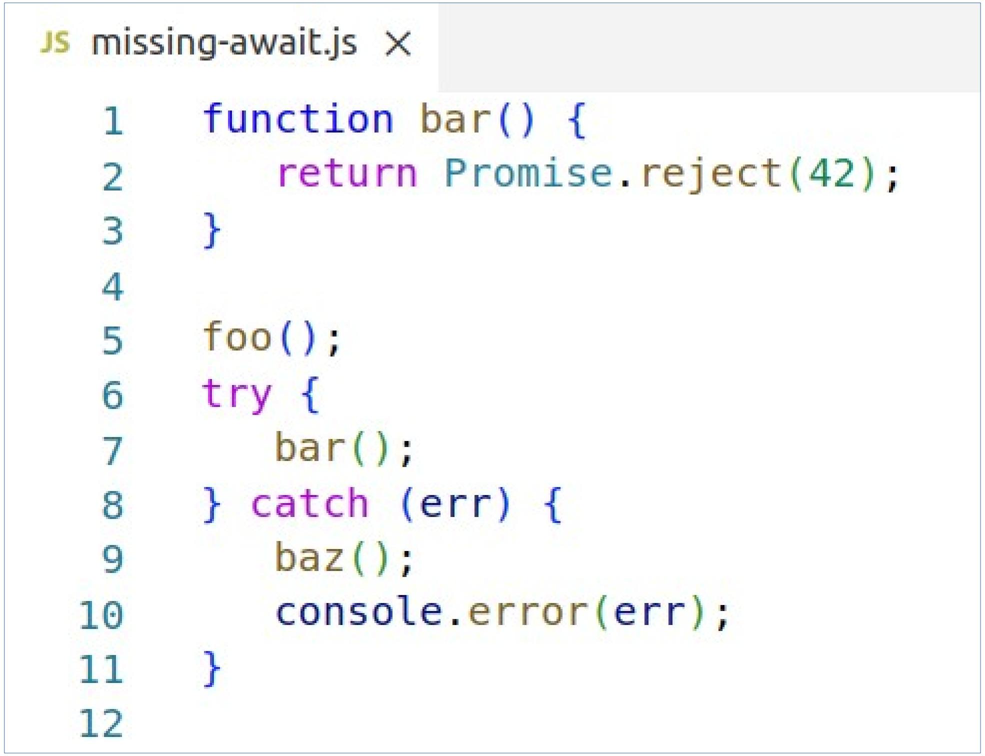


```
foo();
async function bar() {
    await sleep(1);
    await barExec;
}
const barExec = bar();
baz();
await barExec;
// never reach here

```

# Classical bug: missing await





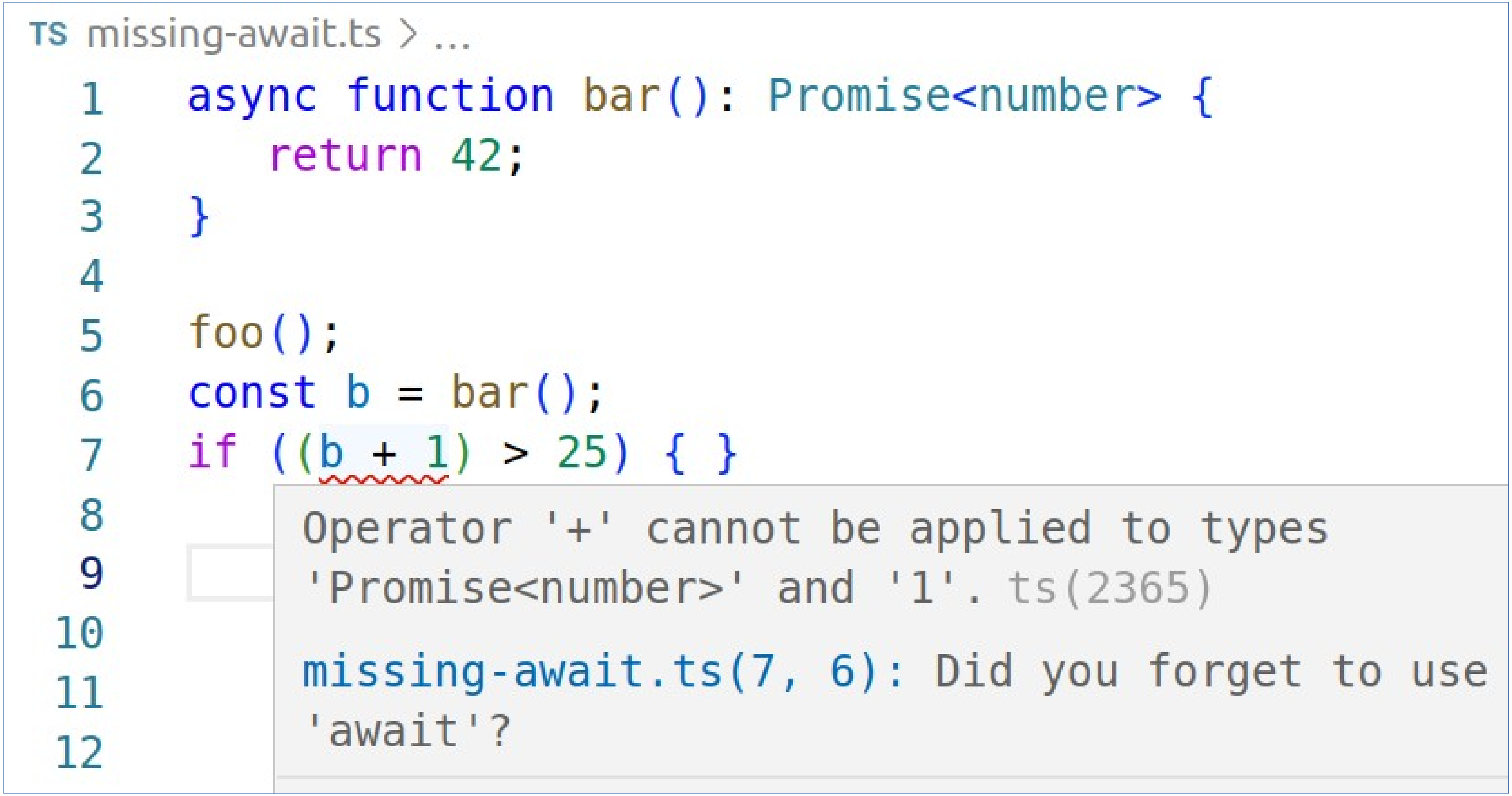
```
function bar() {
    return Promise.reject(42);
}

foo();
try {
    bar();
} catch (err) {
    baz();
    console.error(err);
}

```

# Romeo didn’t wait. So common!

# Typescript to rescue



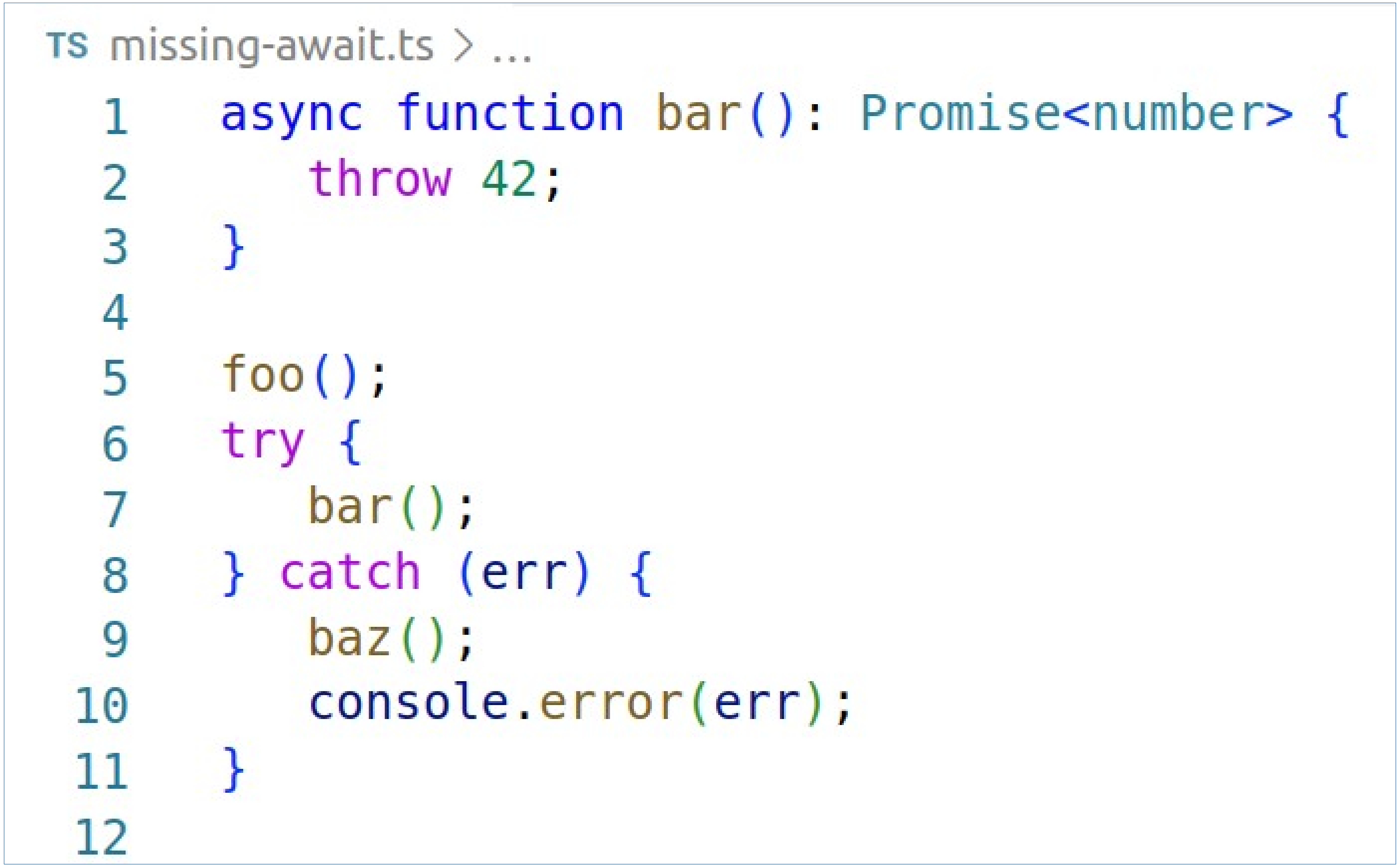
TS missing-await.ts > ...

```
1   async function bar(): Promise<number> {
2       return 42;
3   }
4
5   foo();
6   const b = bar();
7   if ((b + 1) > 25) { }
8
9
10
11
12
```

```
Operator '+' cannot be applied to types
'Promise<number>' and '1'. ts(2365)

missing-await.ts(7, 6): Did you forget to use
'await'?
```

# Without assignment to variable use eslint `@typescript-eslint/no-floating-promises` rule



```
async function bar(): Promise<number> {
   throw 42;
}

foo();
try {
   bar();
} catch (err) {
   baz();
   console.error(err);
}

```

# In Summary

- You can have any logical processes performed by single event loop.
- Await/async is a sane way to structure logical processes.
- Synchronization ~ Ordering of execution
- When computer checks types it helps to catch bugs

# On to the demo:

https://github.com/3n-mb/missing-await-talk-demo