Async Computation

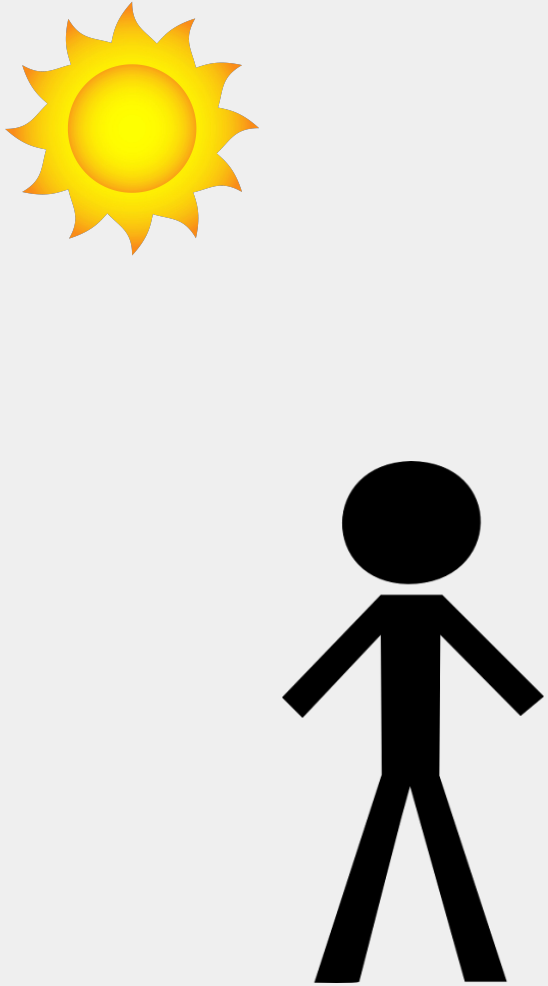
Nicholas Tsao

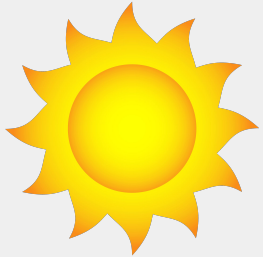
Synchronous

Happens consecutively, one after another

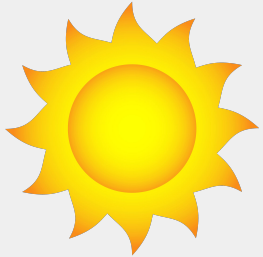
Asynchronous

Happens later at some point in time



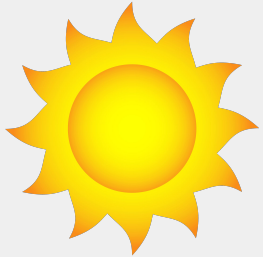



ICE CREAM
FRESHLY MADE



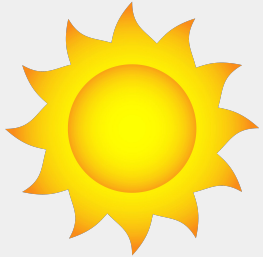

I want 5 snow cones


ICE CREAM
FRESHLY MADE

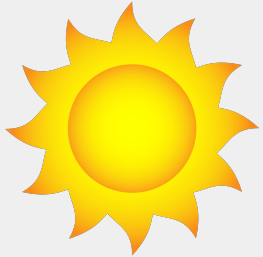



Can I have 5 snow cones?

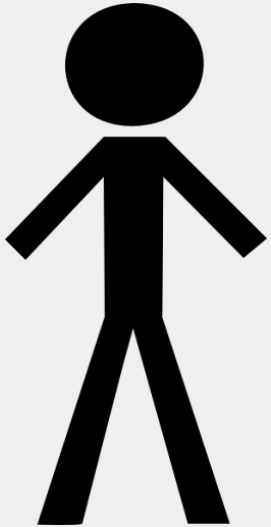

ICE CREAM
FRESHLY MADE

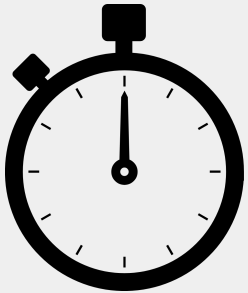


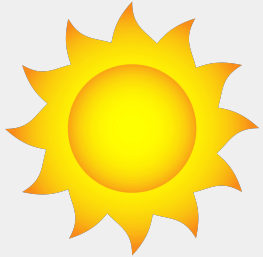
Ok.



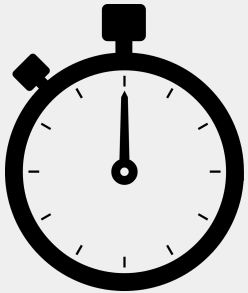

ICE CREAM
FRESHLY MADE

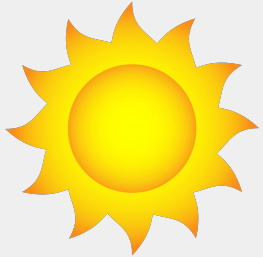




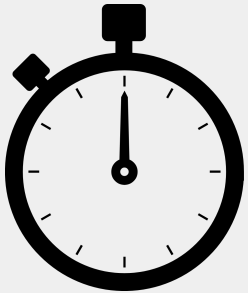


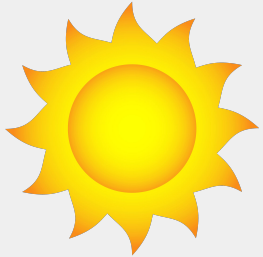

ICE CREAM
FRESHLY MADE



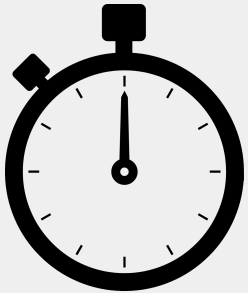


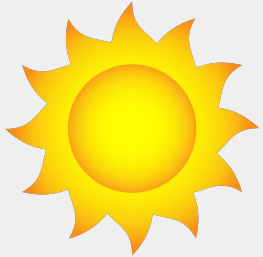

ICE CREAM
FRESHLY MADE



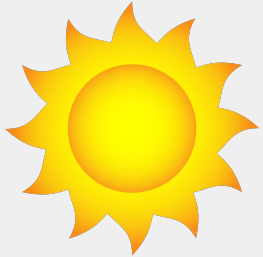



ICE CREAM
FRESHLY MADE

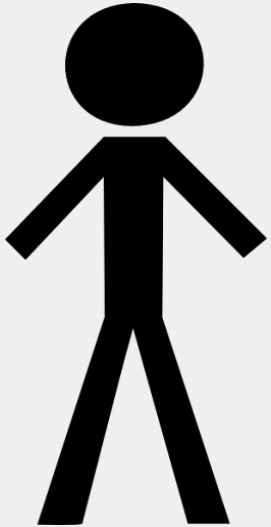




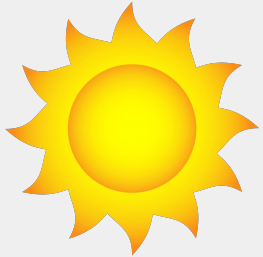

ICE CREAM
FRESHLY MADE



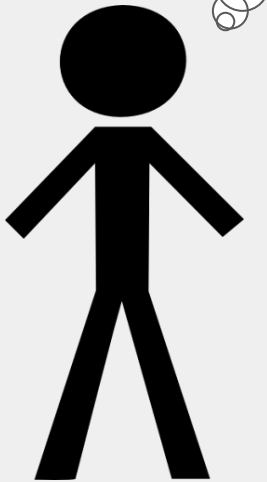

ICE CREAM
FRESHLY MADE

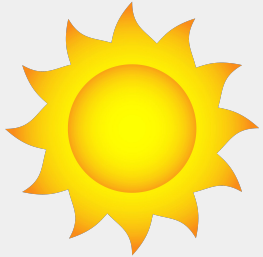


What if there is more than one truck?

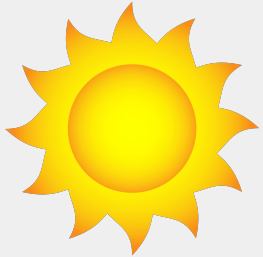


I want 5 snow cones





Can I have 5 snow cones?



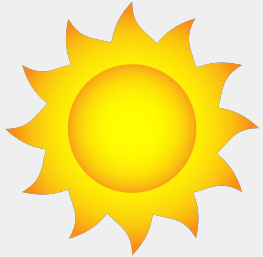

Can I have 5 snow cones?


ICE CREAM
FRESHLY MADE

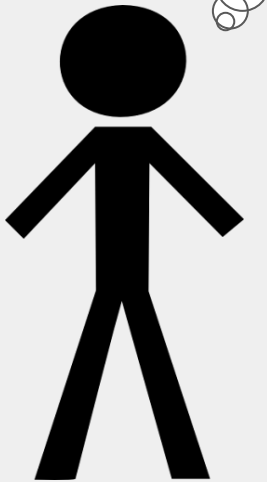

ICE CREAM
FRESHLY MADE

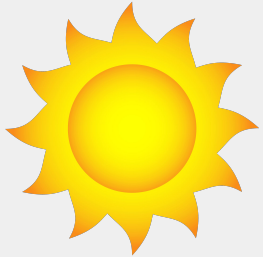

ICE CREAM
FRESHLY MADE

Can we do better?



I want 5 snow cones



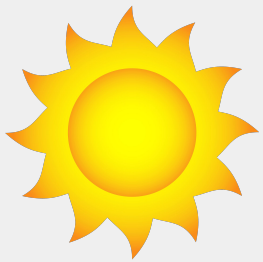



Can I have 2 snow cones?


ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE

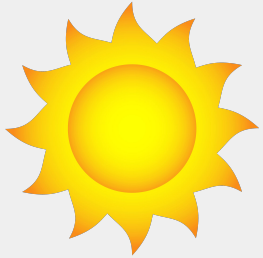


Ok.


ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE

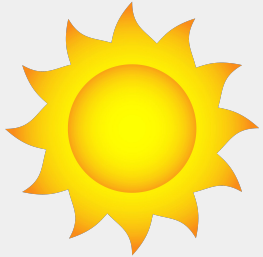

ICE CREAM
FRESHLY MADE



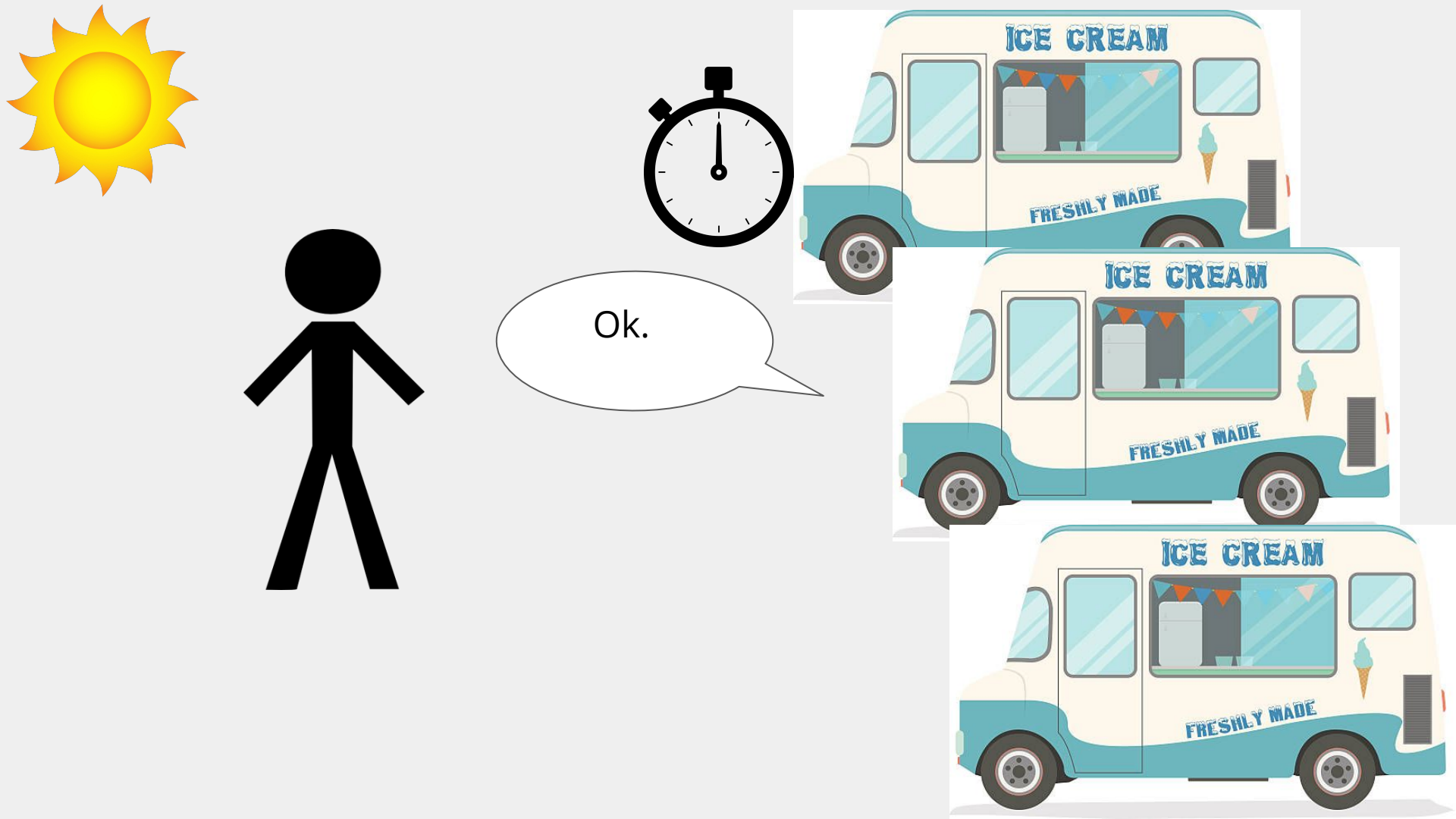

ICE CREAM
FRESHLY MADE

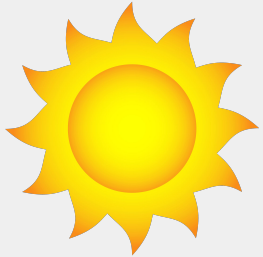

ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE



Can I have 2 snow cones?

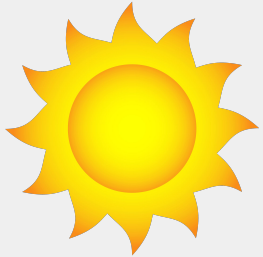

Ok.
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE




ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE

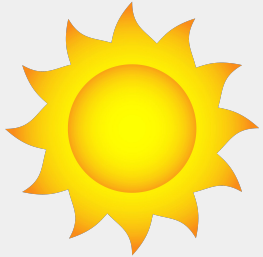



ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE

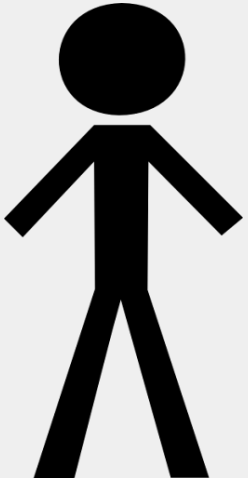

Can I have 1 snow cone?


ICE CREAM
FRESHLY MADE



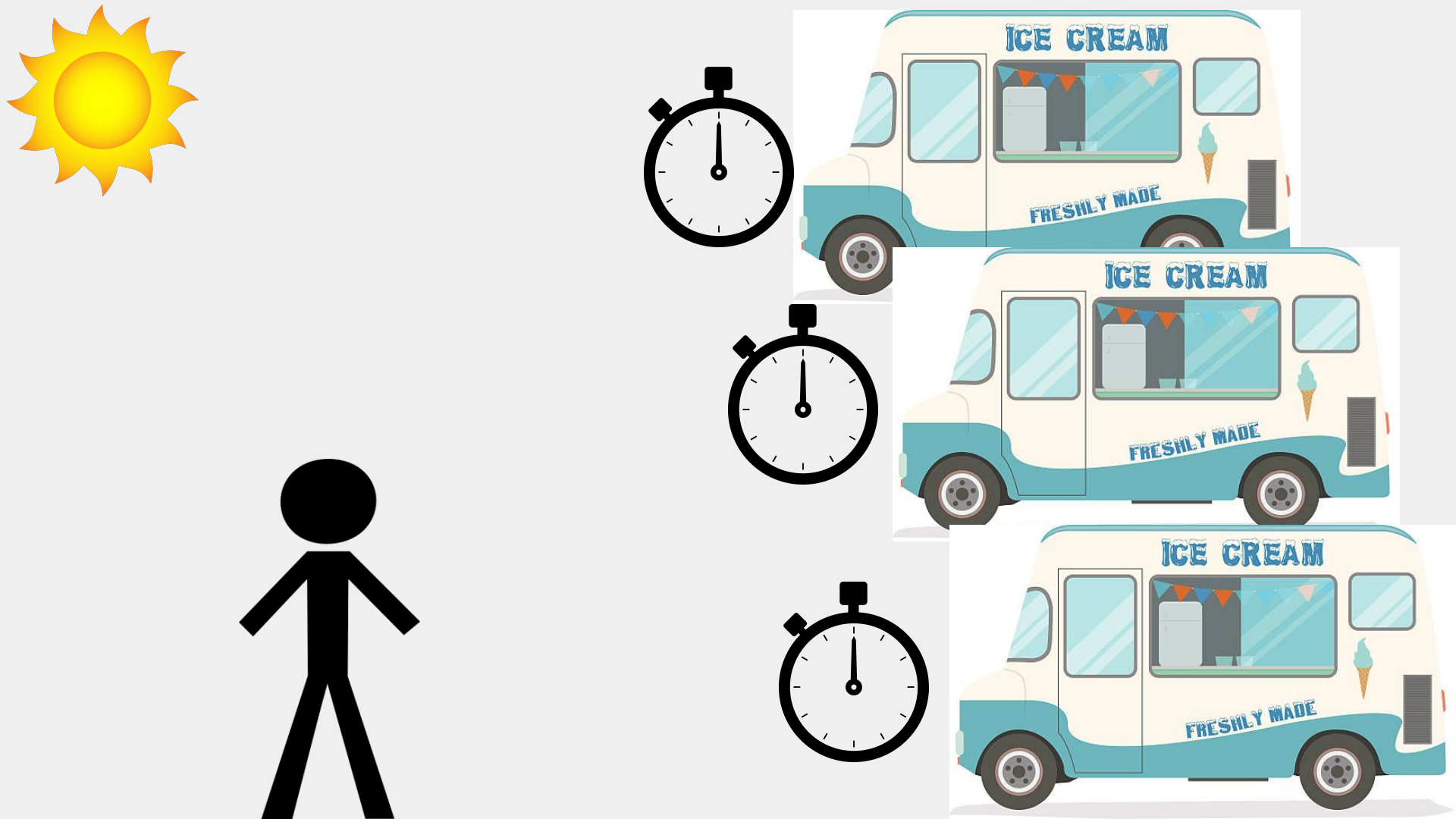

ICE CREAM
FRESHLY MADE

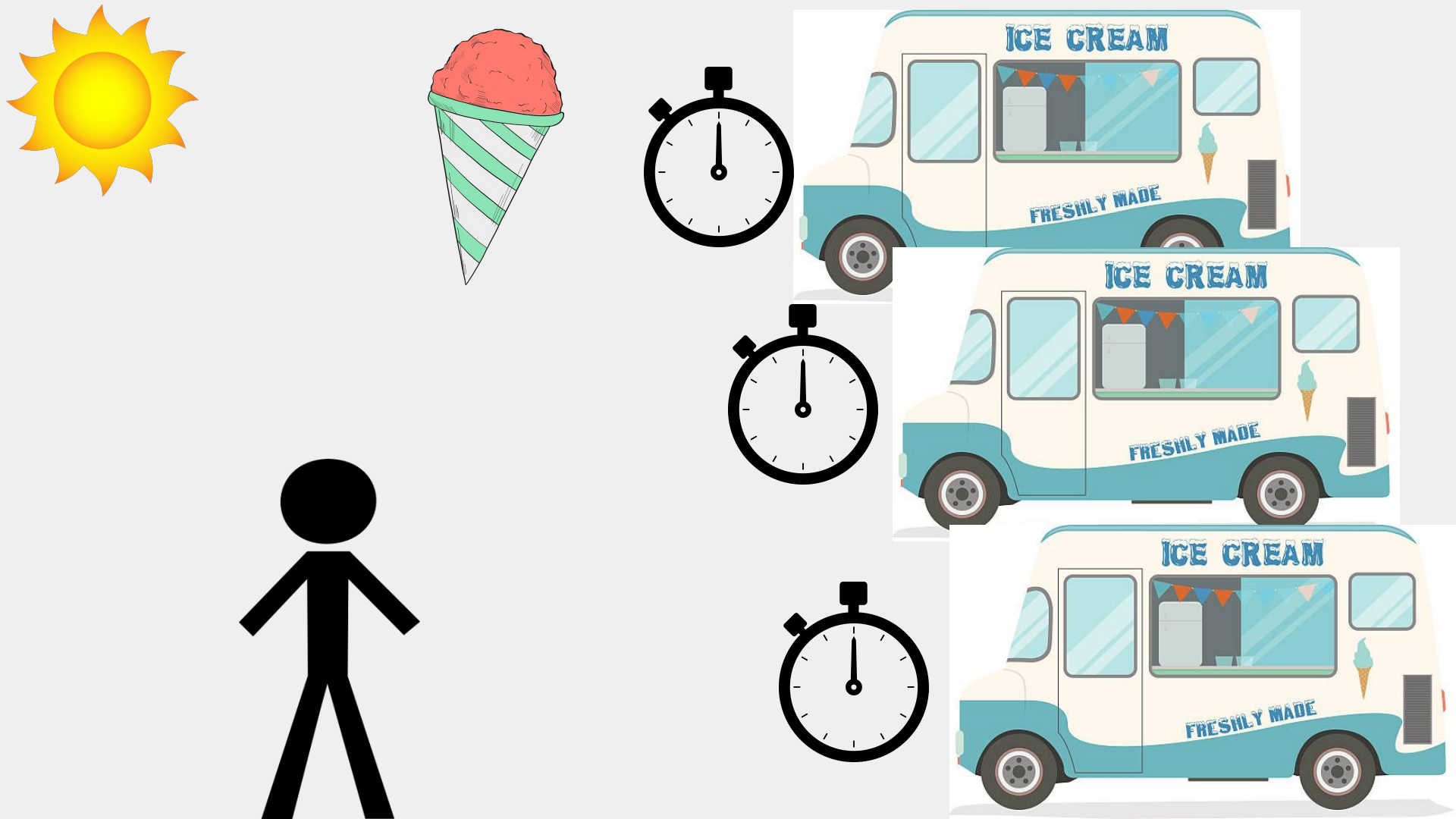

ICE CREAM
FRESHLY MADE

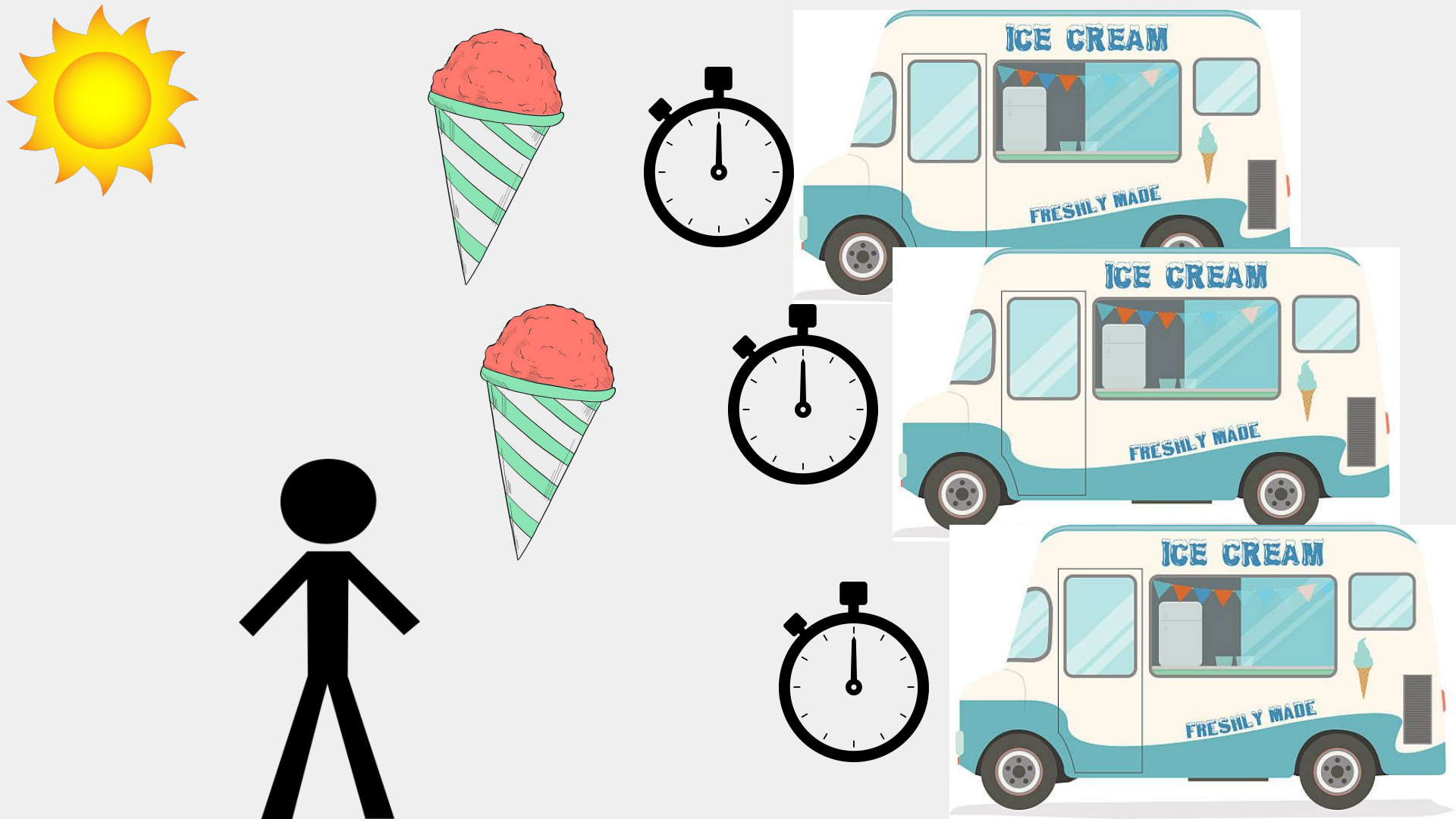


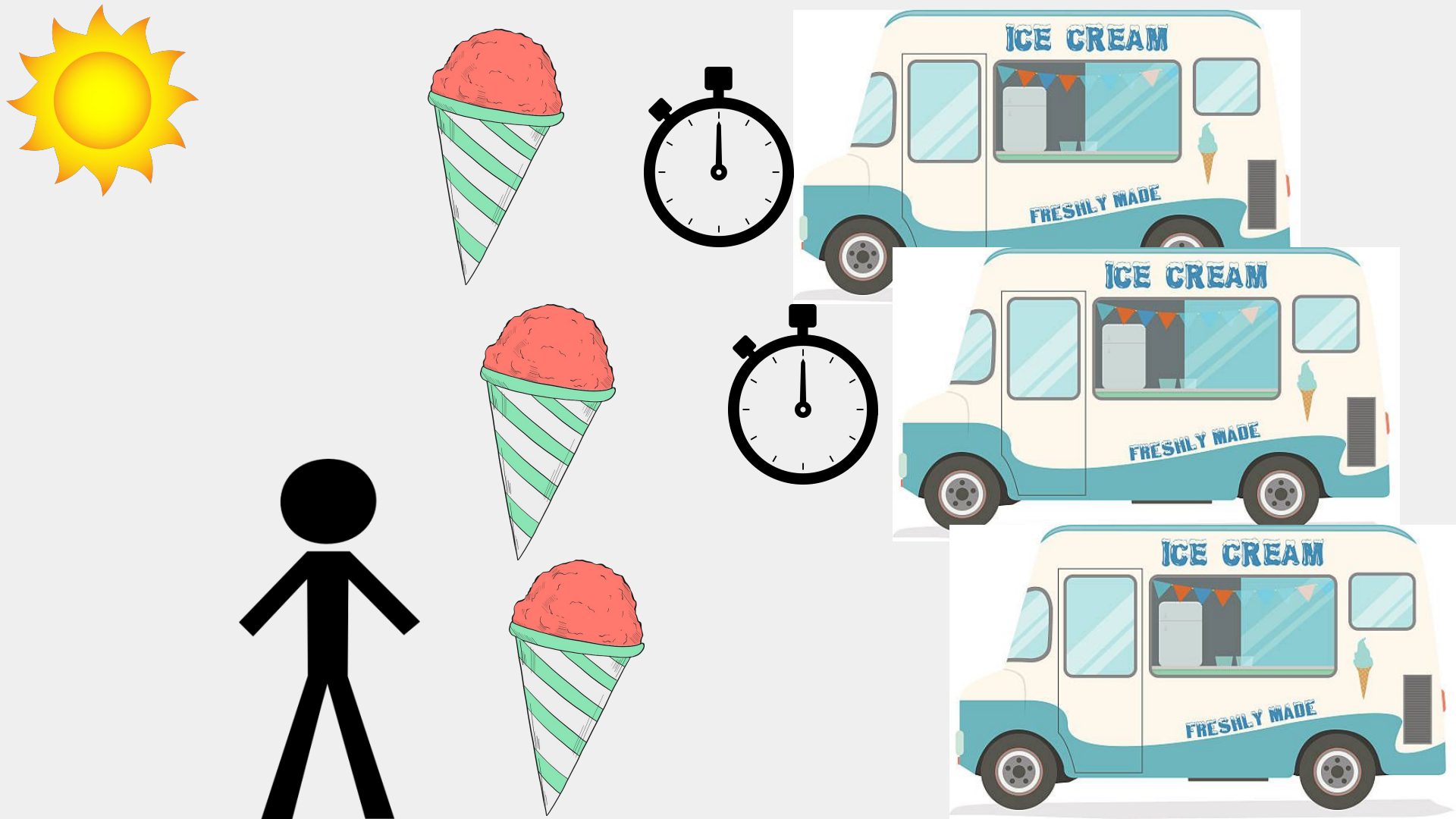
Ok.

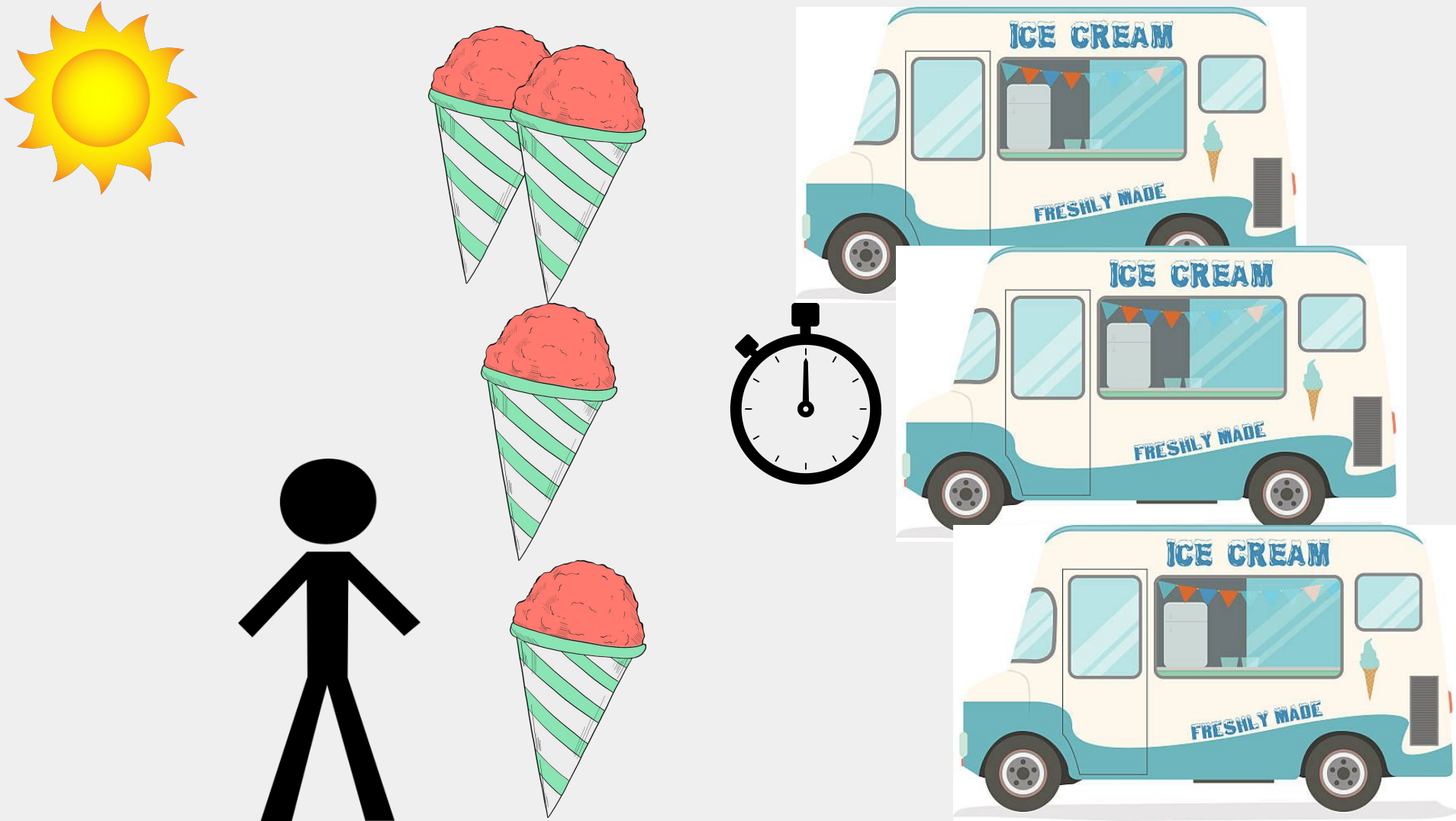

ICE CREAM
FRESHLY MADE

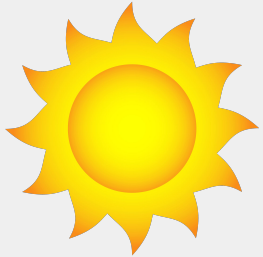

ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE

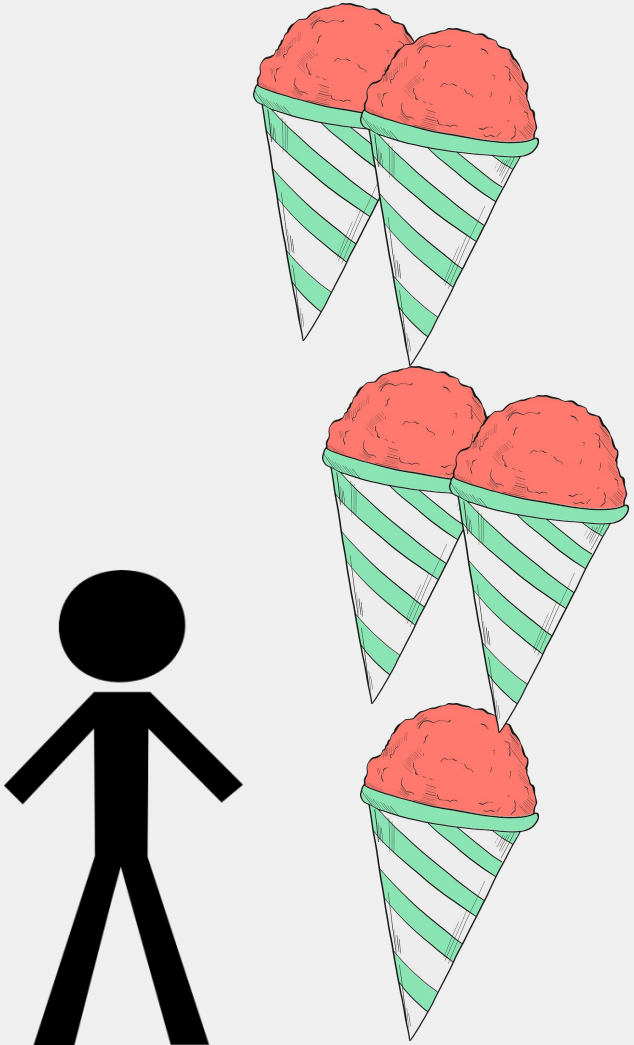

ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE
ICE CREAM
FRESHLY MADE






ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE


ICE CREAM
FRESHLY MADE

Promises

Promises

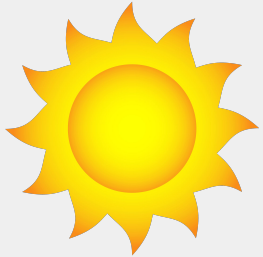
- A promise is an object in JavaScript
- Represents some asynchronous task

Promises exist in three states

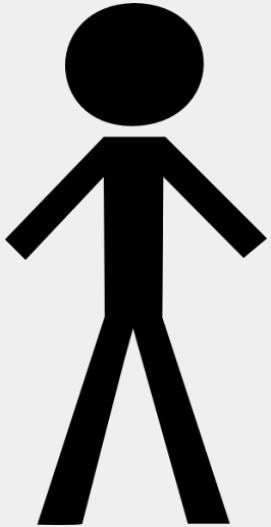
- **Pending** - initial state (neither fulfilled nor rejected)
- **Fulfilled** - the operation completed successfully
- **Rejected** - the operation failed

(Asynchronous function: Make snow cone)

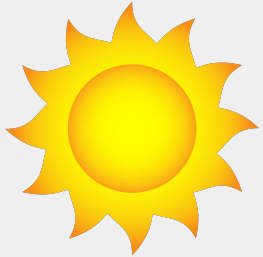
You call an asynchronous function to do some asynchronous task.

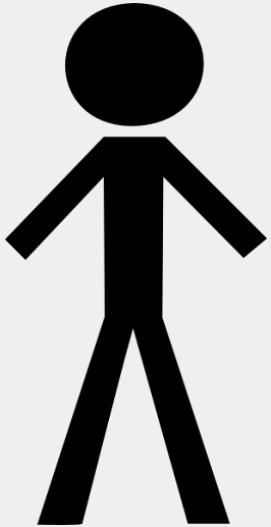


Ok. (I'm working on it.)

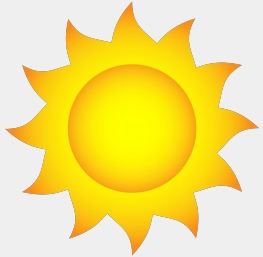


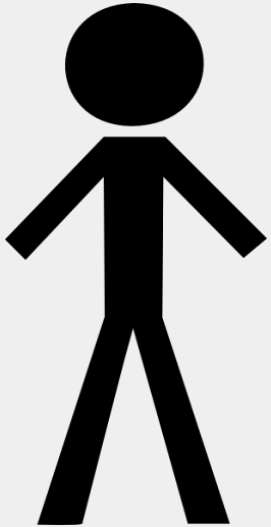
The asynchronous function immediately returns a promise. It "promises" to give you a snow cone later.



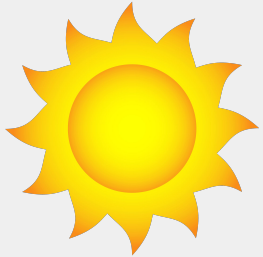


The promise is **pending** (the task is not done yet).



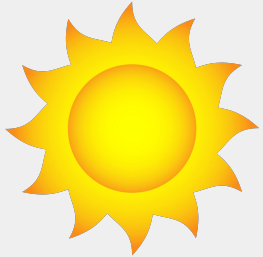


The asynchronous task has finished. The promise has resolved into an actual value (the snow cone). The promise is now **fulfilled**.

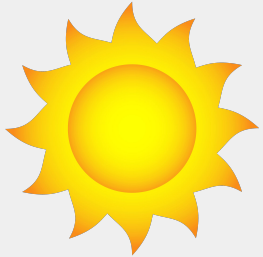


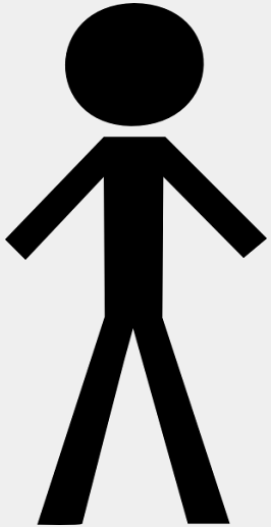

Can I have a snow cone?


ICE CREAM
FRESHLY MADE

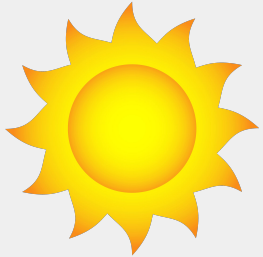


Ok.



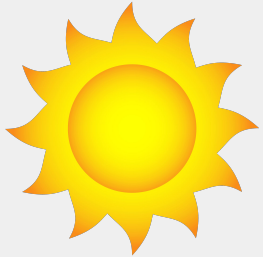


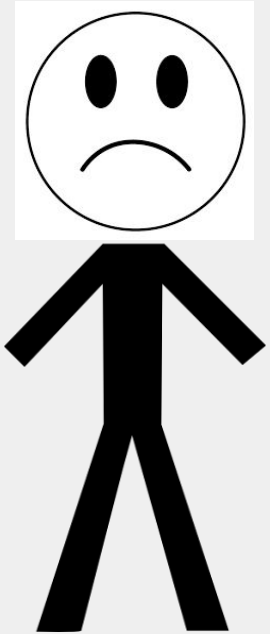
(Promise **pending**)



Sorry, we have no more ice.

The asynchronous function ran into some error. It cannot fulfill the promise. Promise **rejected**.






ICE CREAM
FRESHLY MADE

Promises in JavaScript

What do promises look like?

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  });
}, []);
```

What do promises look like?

```
useEffect(() => {
  const stories = get("/api/stories");
  console.log(stories);
}, []);
```

```
useEffec  const stories: Promise<any>
  const stories = get("/api/stories");
  console.log(stories);
```

```
▶ Promise {<pending>}
```

Handle promises with .then() and .catch()

.then()

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  });
}, []);
```

Once the promise is **fulfilled**, do stuff (call a callback function).
Returns a promise.

.catch()

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  }).catch((err) => {
    console.log("this is so sad: ", err.message);
  });
}, []);
```

Once the promise is **rejected**, do stuff (call a callback function). Returns a promise.

Chaining promises

.then() returns a promise, so we can do .then() again, and again, and again... (same goes for .catch())

```
getPromise().then((value) => {
    console.log("first promise resolved, let's do some stuff");
}).then((value) => {
    console.log("second promise resolved, let's do more stuff");
}).then((value) => {
    console.log("third promise resolved, :)");
}).catch((err) => {
    console.log("oops i am sad now :(")
});
```

Async / Await

Doing stuff with promises

You can’t compute with pending promises

```
const a = slowNumber(9);
const b = slowNumber(10);

console.log(a + b);
```

(slowNumber(x) returns the number x after 1 second)

JS doesn’t wait for the promises to resolve before continuing.

```
[object Promise][object Promise]
```

JS doesn’t know what a and b are when it does this addition. It just sees 2 pending promises.

Await

- You can’t do computation with pending promises
- Wait for the promise to resolve
- Get the value that it resolves to

Using await

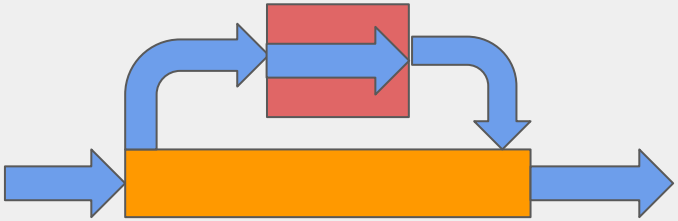
```
const a = slowNumber(9);
const b = slowNumber(10);

console.log(await a + await b);
```

await waits for the promise to **resolve** and uses that value

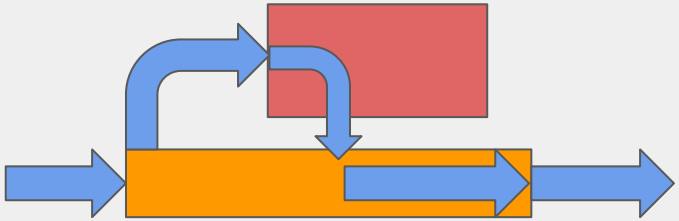
However, if you type this directly into VSCode, it will **NOT** work. It just gives an error when you try to run.

Asynchronous functions

- Functions that return control back to the caller before computation is done



synchronous



asynchronous

Asynchronous functions

- Functions that return control back to the caller before computation is done
- Can be made as a callback function () => {}

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  });
}, []);
```

Asynchronous functions

- Functions that return control back to the caller before computation is done
- Can be made as a callback function () => {}
- **OR** with the **async** keyword
  - Works with function, arrow functions, class methods, etc.

```
async function slowNumber(x) {
    sleep(1000);
    return x;
}
```

Only asynchronous functions can use await

Await returns control back to the caller

Await is like a temporary return that reactivates once the promise it is waiting for has resolved (then it takes control back).

Using await

Notably, the outermost level of our program is **NOT** an async function, so it **CANNOT** use await. However, it will wait to resolve any promises at the end of the program before exiting.

```
async function main() {
    const a = slowNumber(9);
    const b = slowNumber(10);

    console.log(await a + await b);
}
main();
```

This prints 19 (as expected).

Using multiple promises

Many promises

```
1   const promise1 = get('/api/comments', { parent: parentId1 });
2   const promise2 = get('/api/comments', { parent: parentId2 });
3   const promise3 = get('/api/comments', { parent: parentId3 });
4   const promise4 = get('/api/comments', { parent: parentId4 });
5   const promise5 = get('/api/comments', { parent: parentId5 });
6
7   const promises = [promise1, promise2, promise3, promise4, promise5];
```

Promise.all()

```
1   const promise1 = get('/api/comments', { parent: parentId1 });
2   const promise2 = get('/api/comments', { parent: parentId2 });
3   const promise3 = get('/api/comments', { parent: parentId3 });
4   const promise4 = get('/api/comments', { parent: parentId4 });
5   const promise5 = get('/api/comments', { parent: parentId5 });
6
7   const promises = [promise1, promise2, promise3, promise4, promise5];
8
9   Promise.all(promises).then((allResults) => {
10    // All results represents a list with the result of each promise
11  }).catch((err) => {
12    // Catch and report any error
13  });
```

Returns a promise that resolves to array of results of input promises

Promise.race()

```
1   const promise1 = get('/api/comments', { parent: parentId1 });
2   const promise2 = get('/api/comments', { parent: parentId2 });
3   const promise3 = get('/api/comments', { parent: parentId3 });
4   const promise4 = get('/api/comments', { parent: parentId4 });
5   const promise5 = get('/api/comments', { parent: parentId5 });
6
7   const promises = [promise1, promise2, promise3, promise4, promise5];
8
9   Promise.race(promises).then((firstResult) => {
10    // Do something with the first result
11  }).catch((err) => {
12    // Catch and report any error
13  });
```

Returns a promise that fulfills or rejects with the first promise that fulfils or rejects

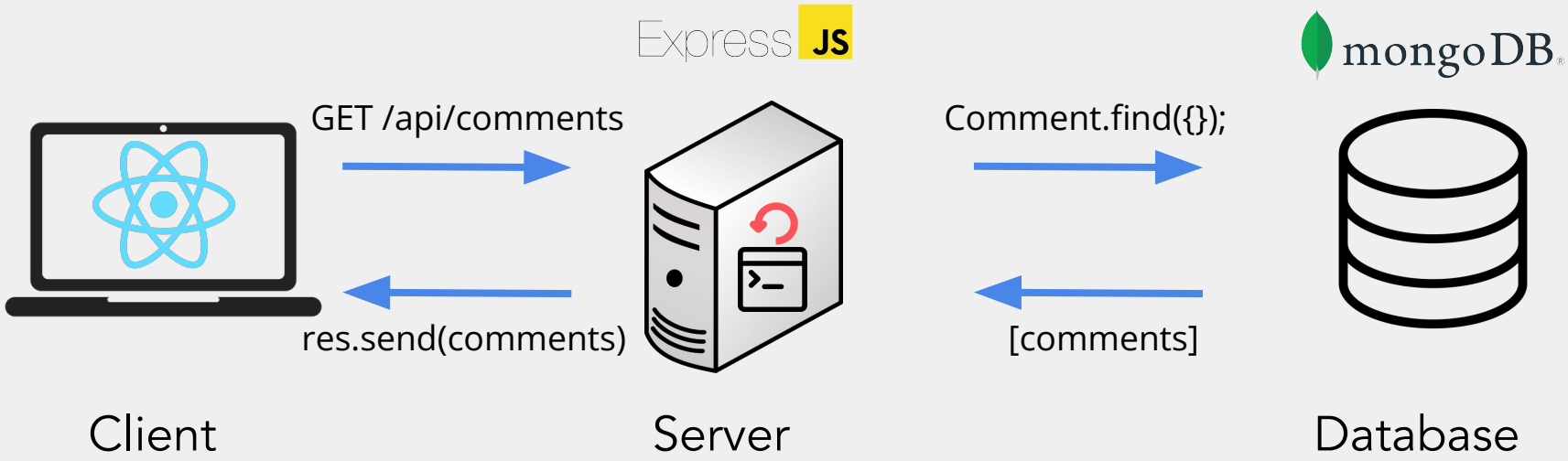
Promise.any()

```
1   const promise1 = get('/api/comments', { parent: parentId1 });
2   const promise2 = get('/api/comments', { parent: parentId2 });
3   const promise3 = get('/api/comments', { parent: parentId3 });
4   const promise4 = get('/api/comments', { parent: parentId4 });
5   const promise5 = get('/api/comments', { parent: parentId5 });
6
7   const promises = [promise1, promise2, promise3, promise4, promise5];
8
9   Promise.any(promises).then((anyResult) => {
10    // Do something with the any result regardless if all others fail
11  }).catch((err) => {
12    // Catch and report any error
13  });
```

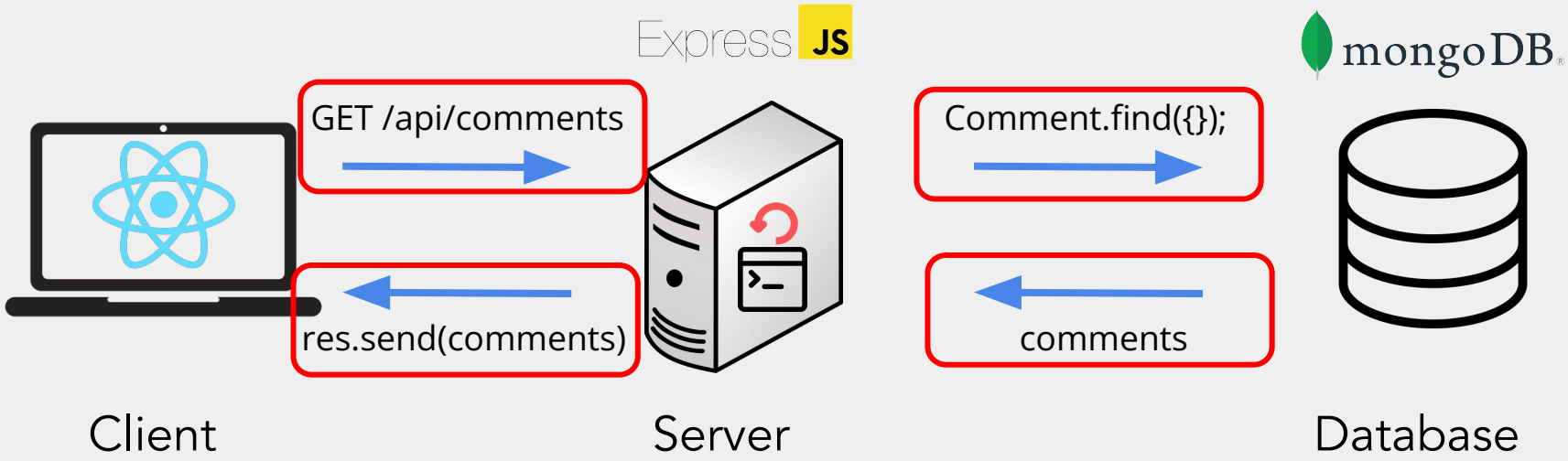
Returns a promise that resolves when any of the input promises fulfills

Why Async?

Communicating between different devices



Unable to guarantee how long it will take to communicate between multiple machines



Client - Server Communication

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  });
}, []);
```

```
router.get("/stories", (req, res) => {
  // empty selector means get all documents
  Story.find({}).then((stories) => res.send(stories));
});
```

Client - Server Communication

(in utilities.js in catbook)

```
export function get(endpoint, params = {}) {
  const fullPath = endpoint + "?" + formatParams(params);
  return fetch(BASE_URL + fullPath)
    .then(convertToJSON)
    .catch((error) => {
      // give a useful error message
      throw `GET request to ${fullPath} failed with error:\n${error}`;
    });
}
```

Client - Server Communication

```
export function post(endpoint, params = {}) {
  return fetch(BASE_URL + endpoint, {
    method: "POST",
    headers: { "Content-type": "application/json" },
    body: JSON.stringify(params),
  })
    .then(convertToJSON) // convert result to JSON object
    .catch((error) => {
      // give a useful error message
      throw `POST request to ${endpoint} failed with error:\n${error}`;
    });
}
```

Client - Server Communication

(in Feed.js)

```
useEffect(() => {
  document.title = "News Feed";
  get("/api/stories").then((storyObjs) => {
    let reversedStoryObjs = storyObjs.reverse();
    setStories(reversedStoryObjs);
  });
}, []);
```

Server - Database Communication

In server.js (in catbook)

```
// connect to mongodb
mongoose
  .connect(mongoConnectionURL, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    dbName: databaseName,
  })
  .then(() => console.log("Connected to MongoDB"))
  .catch((err) => console.log(`Error connecting to MongoDB: ${err}`));
```

Rendering the Front End (React)

Things like setState()

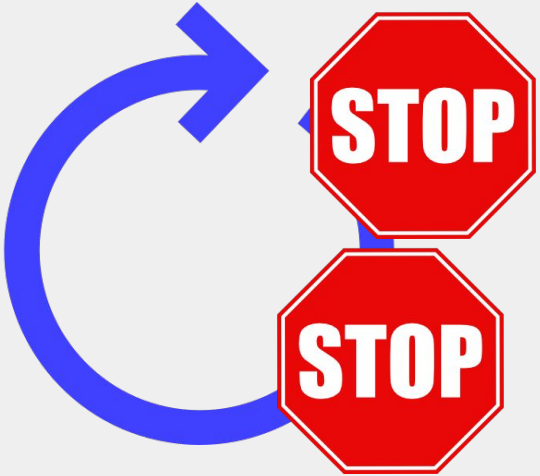
```
const addNewStory = (storyObj) => {
  setStories([storyObj].concat(stories));
};
```

JavaScript Event Loop

Handles everything including things like button presses, inputs, etc.

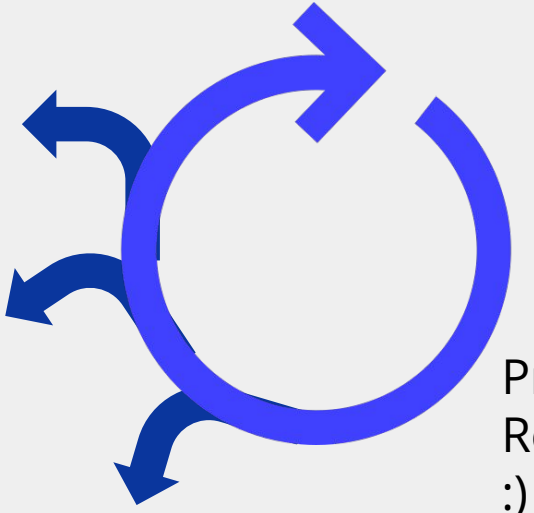
JavaScript Event Loop

Handles everything including things like button presses, inputs, etc.



Synchronous functions

Unresponsive :(



Preserves Responsiveness :)

Asynchronous functions

Background Tasks

You can run background tasks without stopping the user from interacting with the front end.

- Fetching data (e.g. loading new posts on facebook)
- Downloads / uploads
- Run some big computation
- Play music or video (playing music or video on youtube / spotify still lets you click around on other stuff)
- And many more!

Making your own promises

```
import Q from 'q';

async function timeout(milliseconds) {
    const deferred = Q.defer();
    setTimeout(function () {
        deferred.resolve();
        // or we could call deferred.reject(new Error(...)) to reject the promise instead
    }, milliseconds);
    return deferred.promise; // this is the promise that we make
}

timeout(10);
```

Recap

Recap

- Promises have 3 states:
  - **Pending** - initial state (neither fulfilled nor rejected)
  - **Fulfilled** - the operation completed successfully
  - **Rejected** - the operation failed

Recap

Handle promises using .then() and .catch()

- Return promises and can be chained together

```
useEffect(() => {
  get("/api/stories").then((storyObjs) => {
    setStories(storyObjs);
  }).catch((err) => {
    console.log("this is so sad: ", err.message);
  });
}, []);
```

Recap

- Use **await** keyword to wait for promises to resolve
- Use **async** keyword to define asynchronous functions
- You MUST wrap every **await** within an **async** function

Recap

Use async when we don’t know how long something will take:

- Client-server communication
- Server-database communication
- React Front End
- Background Tasks

Make your own promises with Q.deferred

Thanks!

Sponsor Lectures at 1pm: Cresicor and Meta

Come to get some FREE weblab stickers!

Cresicor is IN PERSON

Meta is on zoom

(you might win $50)

(FOUR winners today...)