Asynchronous

Processing

in Web Development

Abraham N. Aldaco-Gastélum, Ph.D.

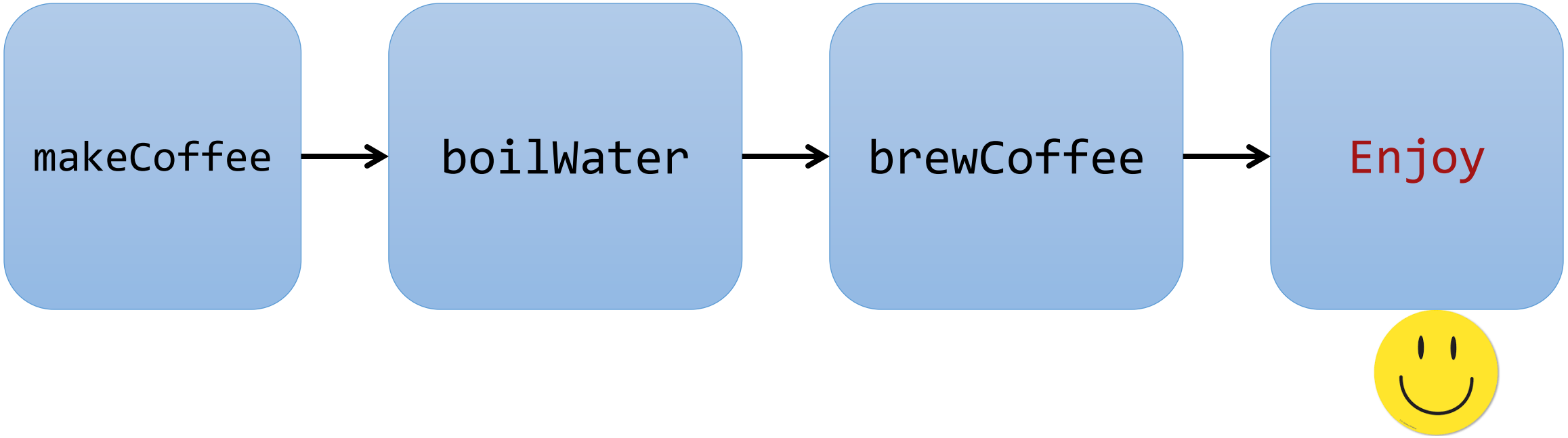
Synchronous Processing

Synchronous Processing

Before showing Asynchronous Processing,

We want to review a simpler and more typical situation in programming processing which is Synchronous processing.

Let’s think in the next process :



In JavaScript, we can prepare coffee :

```javascript
function boilWater() {
    console.log("Boiling water...");
    return "Hot water";
  }

  function brewCoffee(water) {
    console.log(`Brewing coffee with ${water}`);
    return "Coffee";
  }

  function makeCoffee() {
    const water = boilWater();

    const coffee = brewCoffee(water);

    console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
```

In JavaScript, we can prepare coffee :

```javascript
function boilWater() {
    console.log("Boiling water...");
    return "Hot water";
  }

  function brewCoffee(water) {
    console.log(`Brewing coffee with ${water}`);
    return "Coffee";
  }

  function makeCoffee() {
    const water = boilWater();

    const coffee = brewCoffee(water);

    console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
```

Try the code in a browser:

Demo Class University of Wisconsin - Platt

127.0.0.1:53283/index.html

T | Imported From Firefox | ISU | Jobs | English | ComS127 | Weather | ComS/SE319

University of Wisconsin - Platteville

Elements | Console | Sources

top | Filter | Default levels

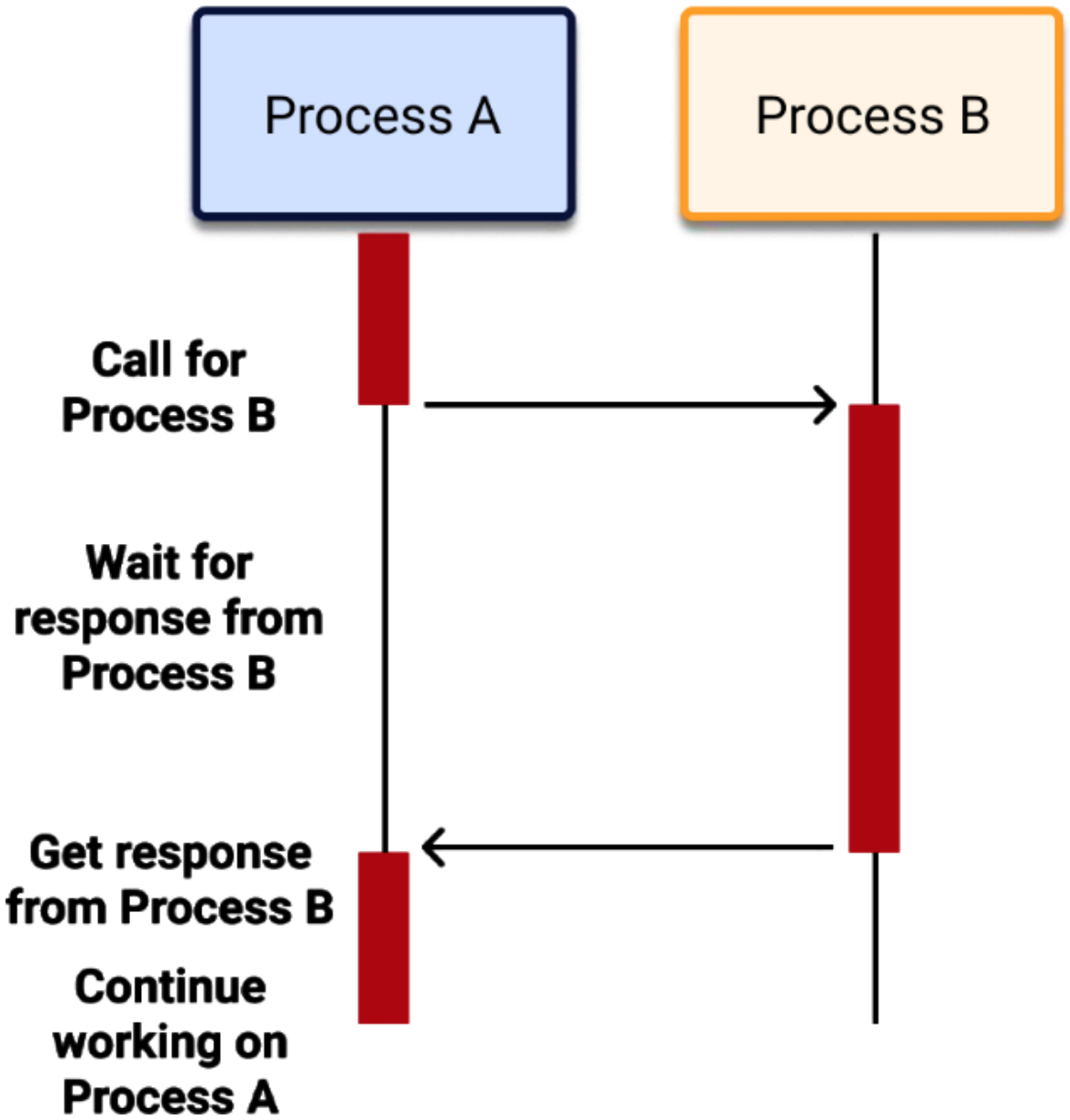
No Issues

```
> function boilWater() {
    console.log("Boiling water...");
    return "Hot water";
  }

  function brewCoffee(water) {
    console.log(`Brewing coffee with ${water}`);
    return "Coffee";
  }

  function makeCoffee() {
    const water = boilWater();
    const coffee = brewCoffee(water);
    console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
```

```
Boiling water...                    VM159:2
Brewing coffee with Hot water       VM159:7
Enjoy your Coffee                   VM159:14
<· undefined
>
```

Analyze output:

Synchronous Processing



In our program there are several part of code that waits their turn…

```
function boilWater() {
    console.log("Boiling water...");
    return "Hot water";
  }

  function brewCoffee(water) {
    console.log(`Brewing coffee with ${water}`);
    return "Coffee";
  }

  function makeCoffee() {
    const water = boilWater();

    const coffee = brewCoffee(water);

    console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
```

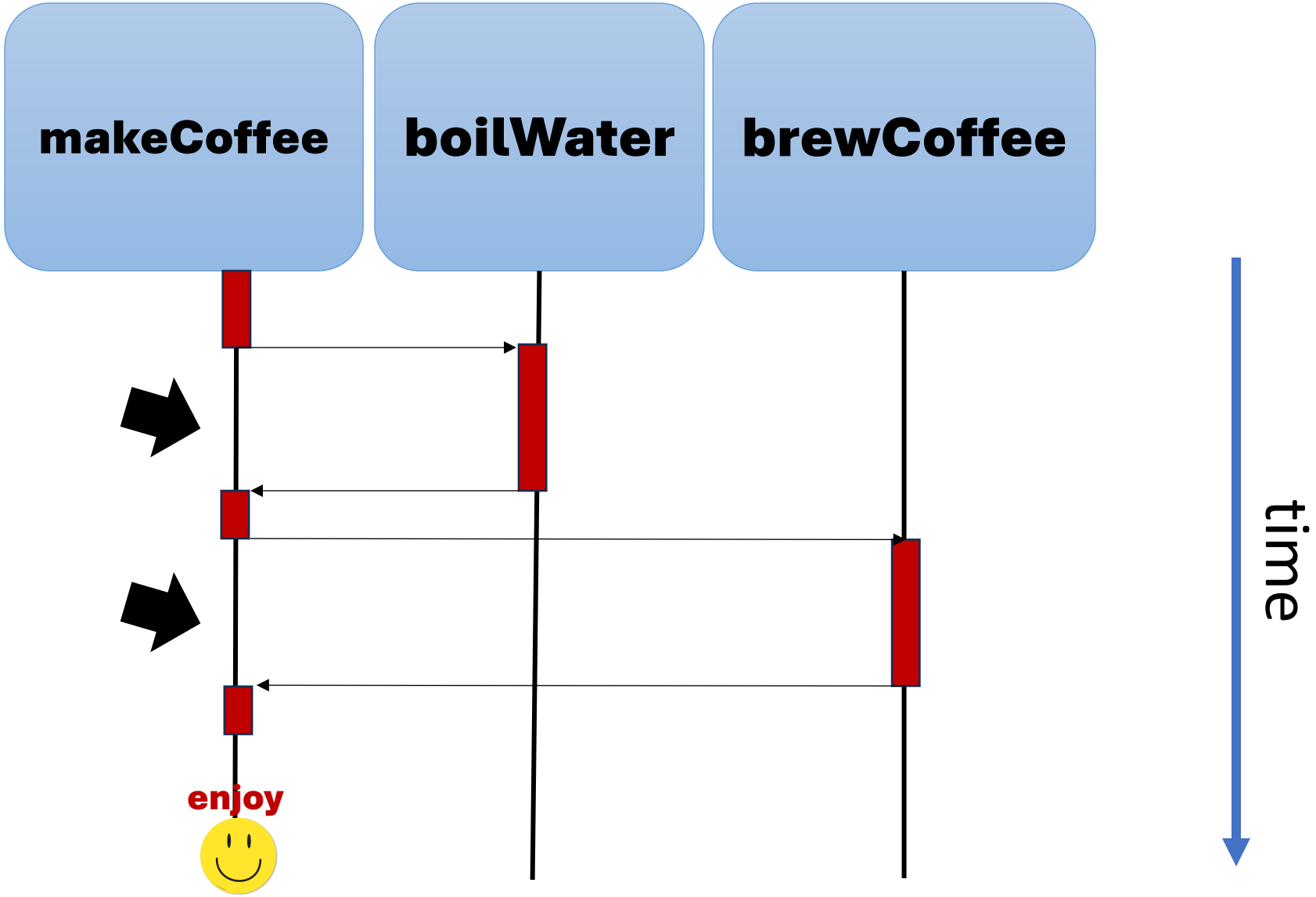
2

console.log() waits to brewCofee() to complete

1

brewCoffee(water) waits to boilWater() to complete.

Here, it is very evident that `brewCoffee()` waits for `boilWater()`



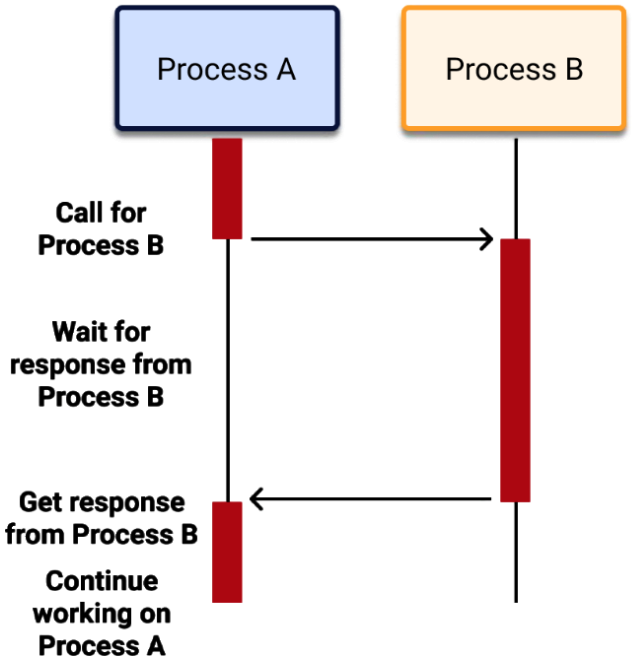
Is synchronous processing good or bad?

It is neither bad nor good,

It is necessary when operations depend on the result of the previous step

In Synchronous processing :

What happens if one of the functions takes too long? How would it affect the program?



Simple JavaScript example with two functions, where one function depends on the result of the previous function.

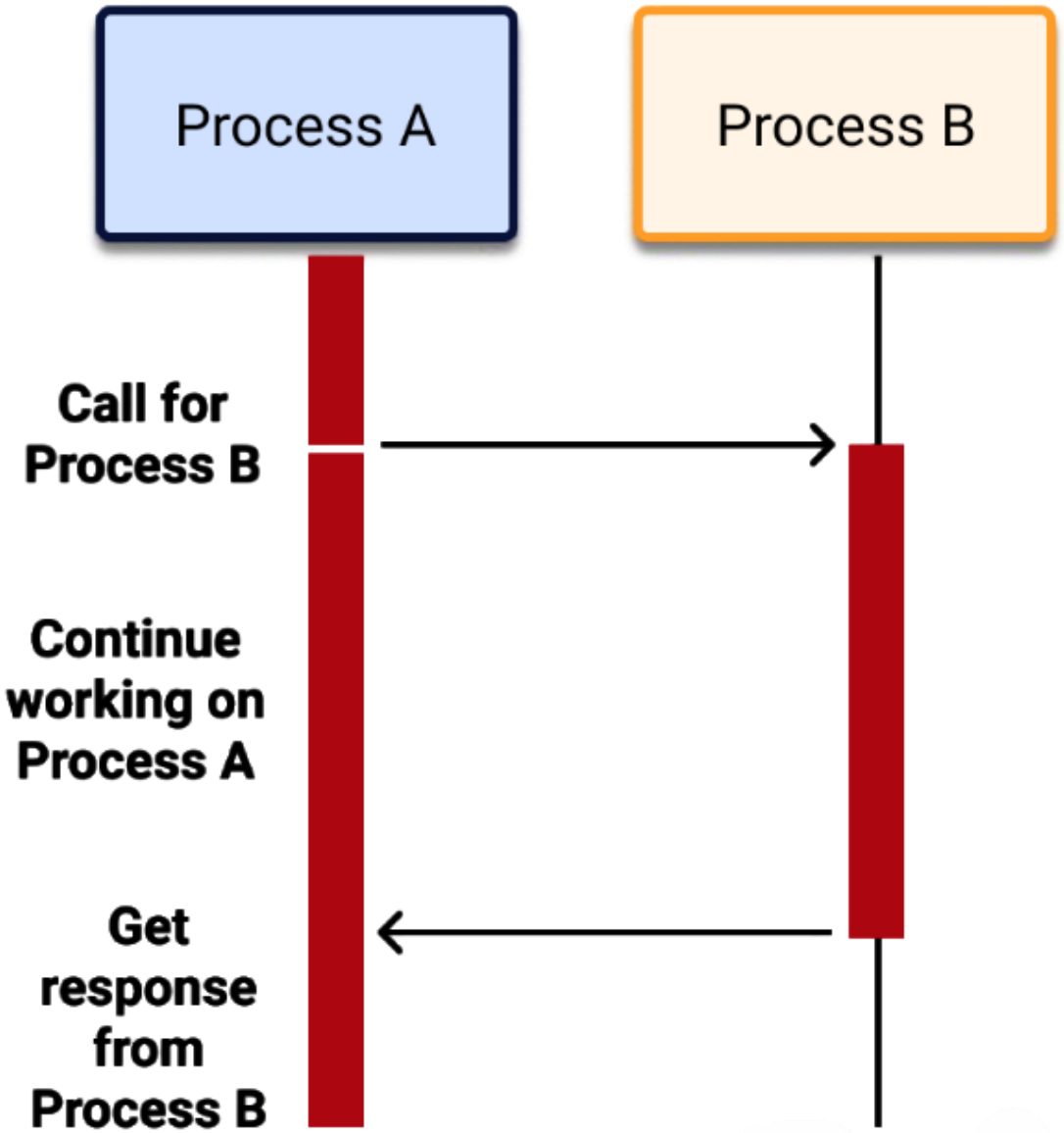
Synchronous Processing

There are situations where one process must wait some intermediate result, but we can continue the execution of other processes (non blocking processing)

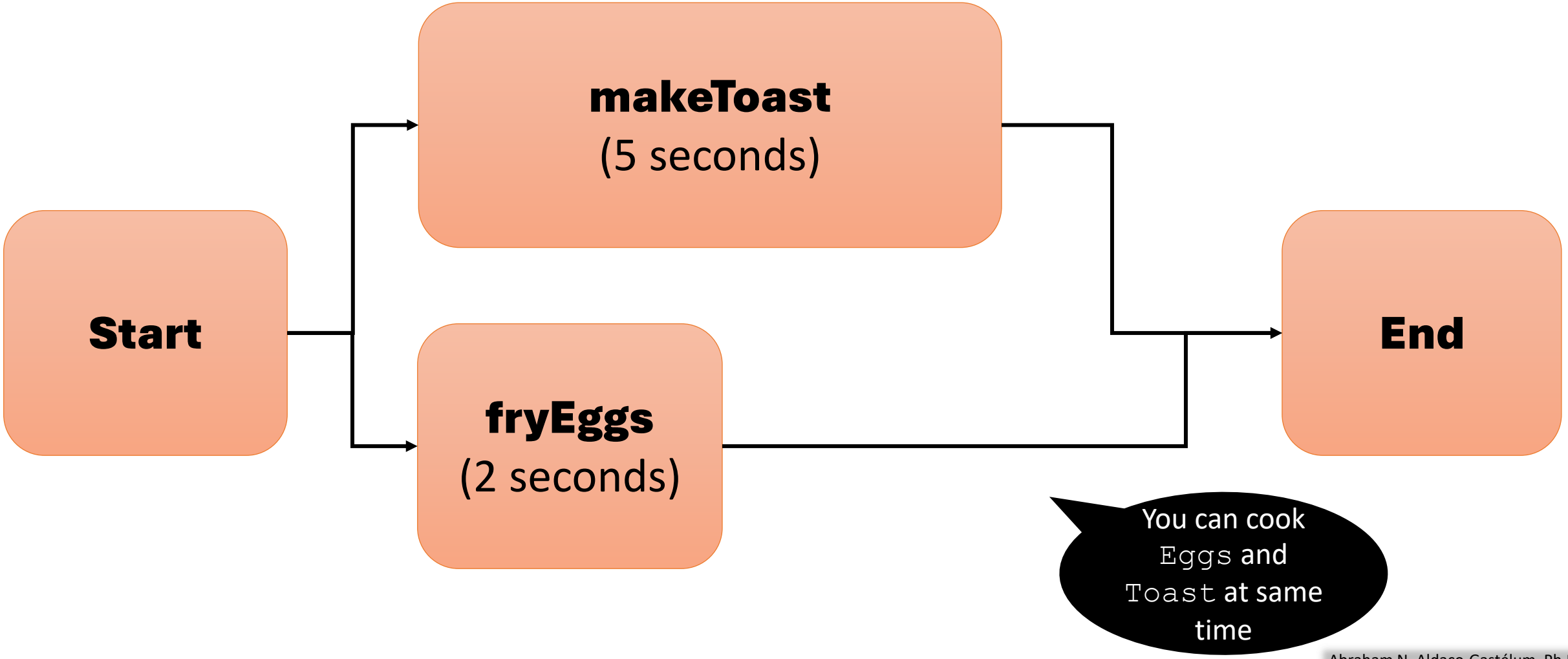
Asynchronous Processing

Asynchronous Processing

Observe what is happening with `Process A` while `Process B` is running.



Let’s simulate two process that can execute without dependency between them:

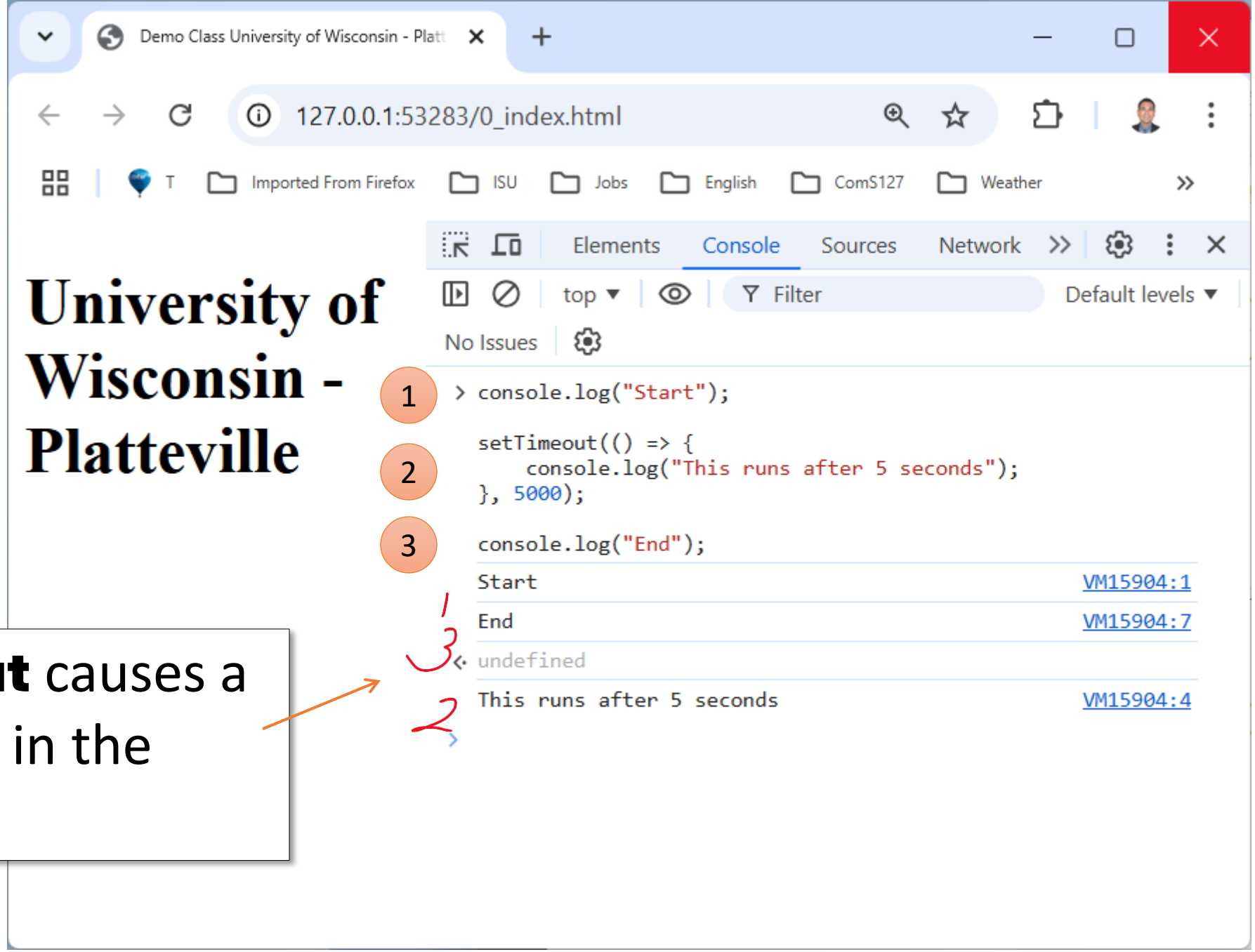


With the help of `setTimeout` we can simulate **delay** in execution :

```
1  console.log("Start");

   setTimeout(() => {
2      console.log("This runs after 5 seconds");
   }, 5000);

3  console.log("End");
```

```
liveserver
python -m http.server 8888
```

Try code in a browser:



setTimeout causes a 5 secs delay in the execution :

Simulate **makeToast**() and **fryEggs**()running simultaneously using setTimeout() :

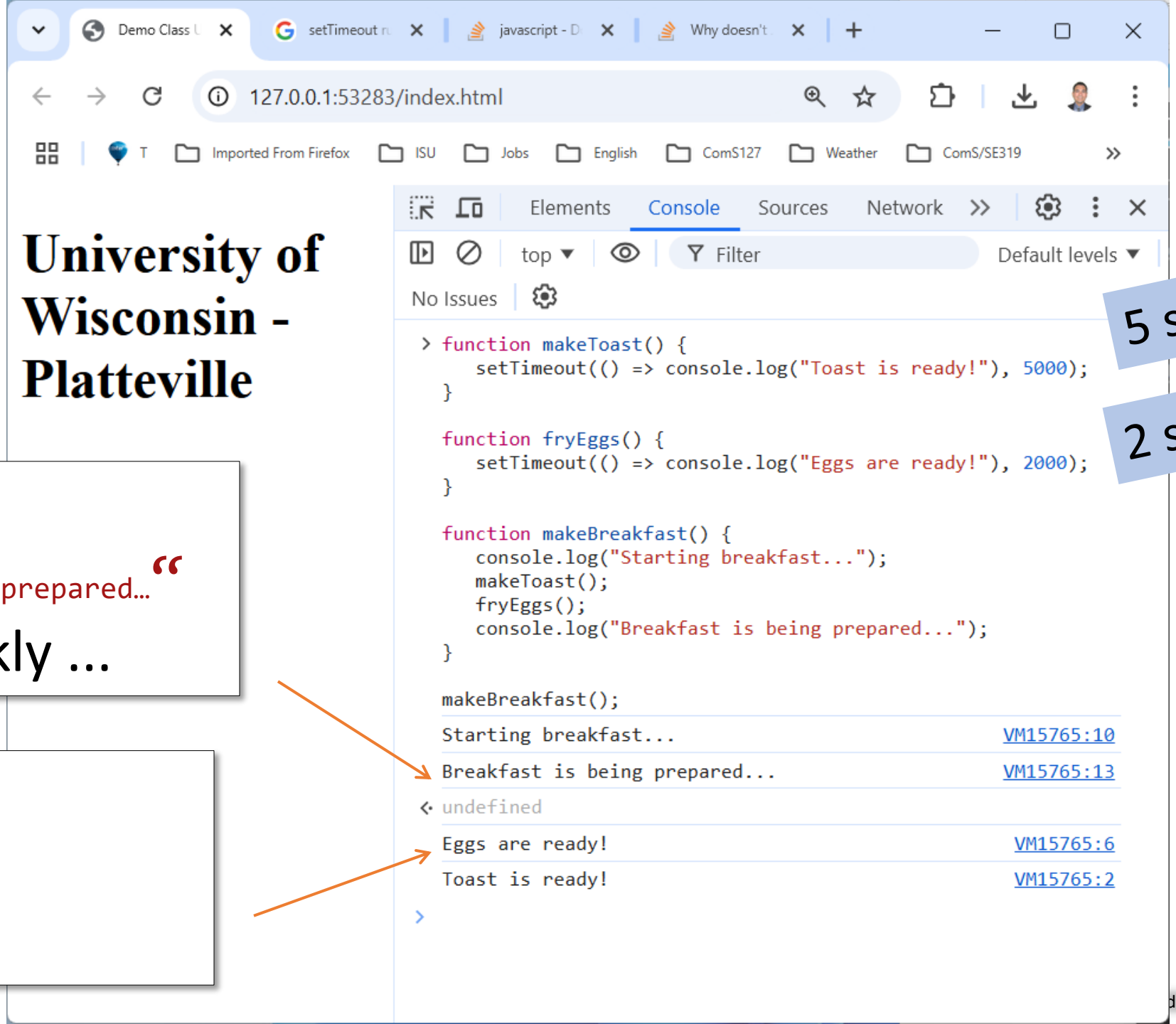
```
function makeToast() {
    setTimeout(() => console.log("Toast is ready!"), 5000);
}
```

5 seconds

```
function fryEggs() {
    setTimeout(() => console.log("Eggs are ready!"), 2000);
}
```

2 seconds

Try code in a browser:



5 seconds

2 seconds

Observe that "`breakfast is being prepared…`" is shown quickly ...

and "`fryEggs()`" finishes first.

```
> function makeToast() {
    setTimeout(() => console.log("Toast is ready!"), 5000);
  }

  function fryEggs() {
    setTimeout(() => console.log("Eggs are ready!"), 2000);
  }

  function makeBreakfast() {
    console.log("Starting breakfast...");
    makeToast();
    fryEggs();
    console.log("Breakfast is being prepared...");
  }

  makeBreakfast();
  Starting breakfast...                    VM15765:10
  Breakfast is being prepared...           VM15765:13
<· undefined
  Eggs are ready!                          VM15765:6
  Toast is ready!                          VM15765:2
>
```

5 seconds

2 seconds

Here, cooking **Eggs** and **Toast** `simultaneously` is not a problem …

It is delicious !

Situation

Scenario :

If you need to download a large volume of data from a server, which can take a considerable amount of time, it would be inefficient for your program or function to freeze while waiting for the data to be fetched.

Instead, it is common practice to run the **fetching** operation in the background.

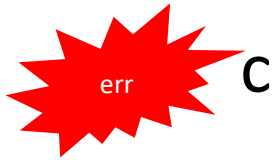
Situation :

Hypothetical situation :

If **processA**() is in charge to load the data from an external source,

and **processB**() in charge of execute over the data (filter, sort, select, etc),

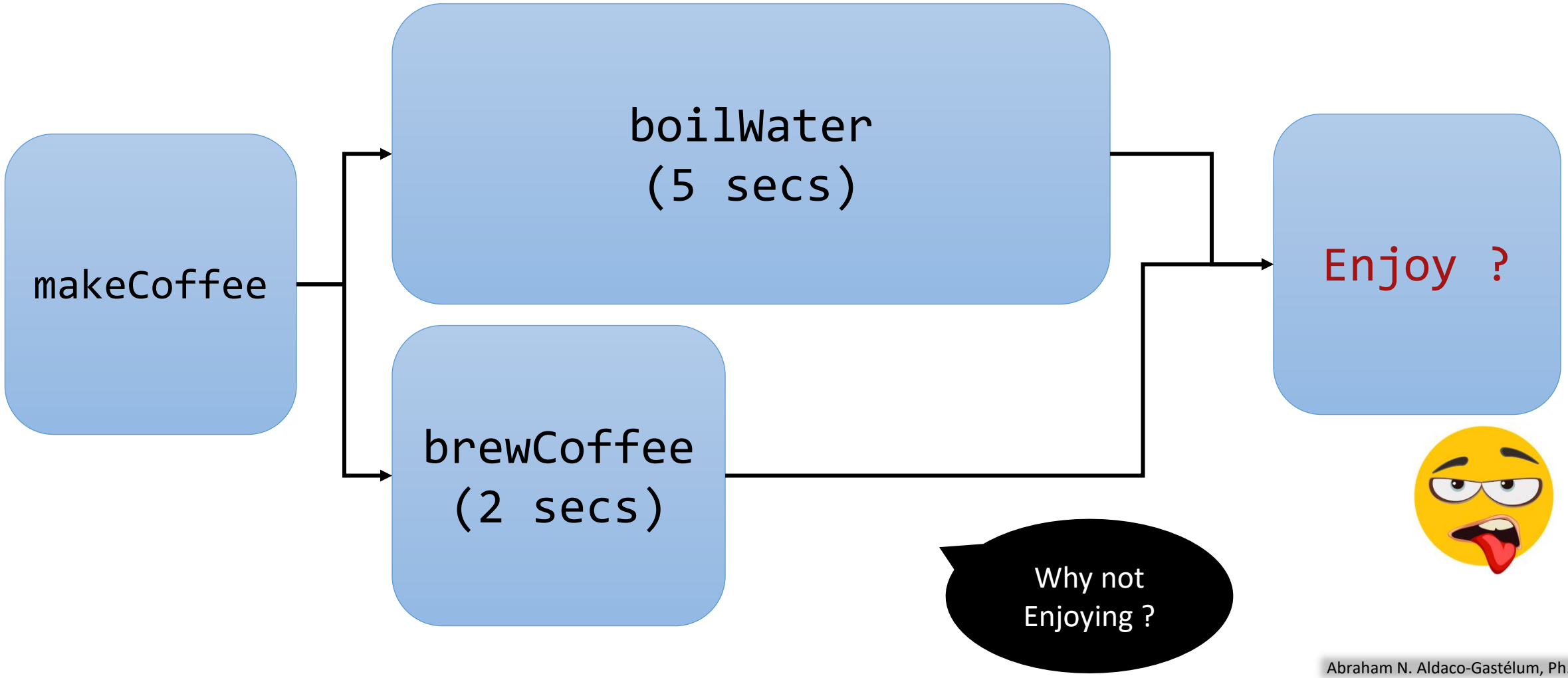
but **processB**() executes before **processA**() concludes,

an 

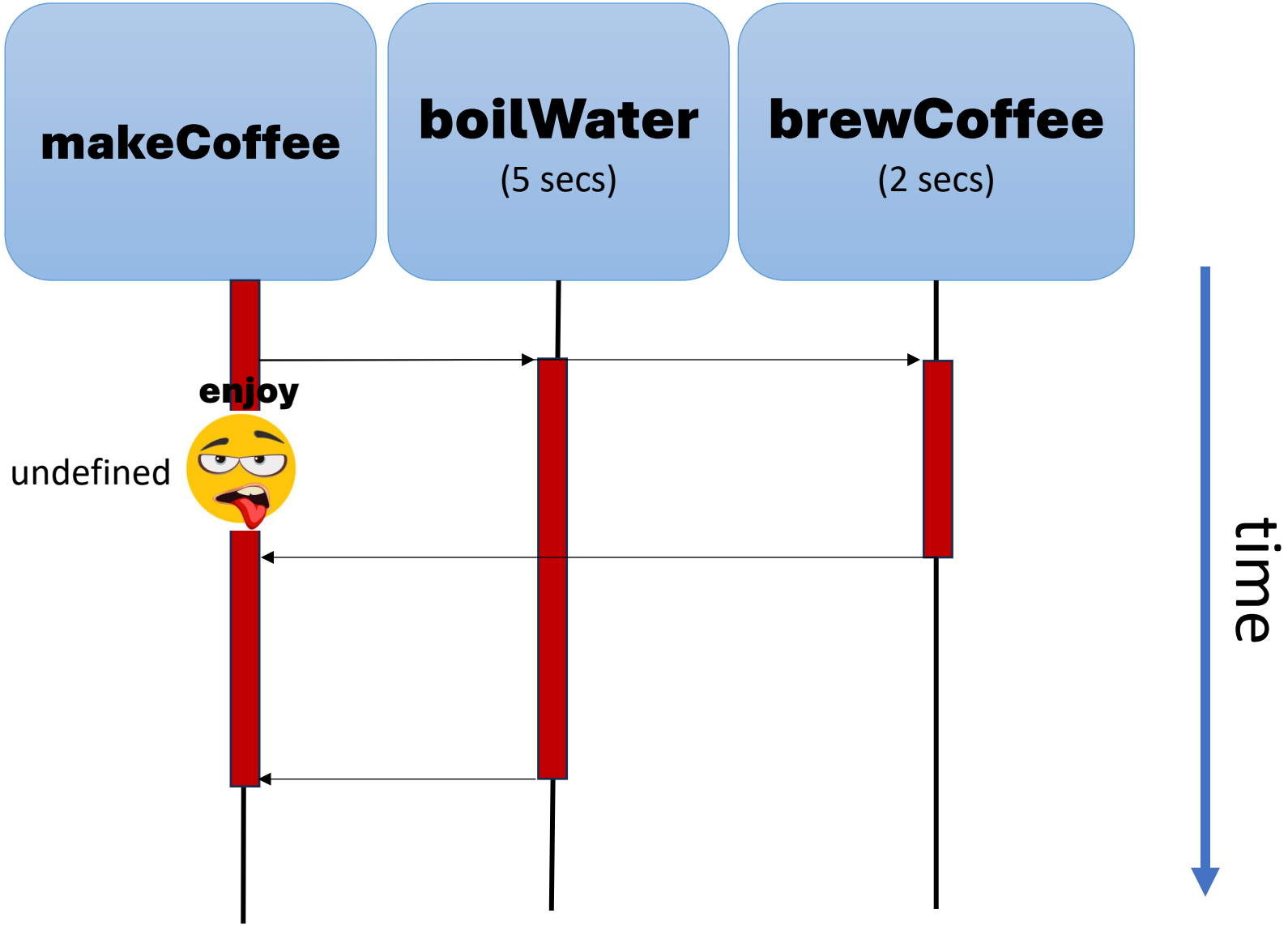
could occur.

Consider our previous example preparing coffee:

This time, we execute the processes **boilWater**() and **brewCoffee**() Asynchronously :



Now, `brewCoffee()` finishes before `boilWater()` and the coffee is not enjoyed .



The output

Try code in a browser:

Demo Class University of Wisconsin - Platt

127.0.0.1:53283/0_index.html

Imported From Firefox | ISU | Jobs | English | ComS127 | Weather | ComS/SE319

University of Wisconsin - Platteville

Elements | Console | Sources

top | Filter | Default levels

No Issues

```
> function boilWater() {
      console.log("Boiling water...");
      setTimeout(() => {
          console.log("Hot water")
          return "Hot water";
      }, 5000);
    }

  function brewCoffee(water) {
      console.log(`Brewing coffee with ${water}`);
      setTimeout(() => {
          console.log("Coffee")
          return "Coffee";
      }, 2000);

  }

  function makeCoffee() {
      const water = boilWater();
      const coffee = brewCoffee(water);
      console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
  Boiling water...                    VM219:2
  Brewing coffee with undefined       VM219:10
  Enjoy your undefined                VM219:21
<· undefined
  Coffee                              VM219:12
  Hot water                           VM219:4
>
```

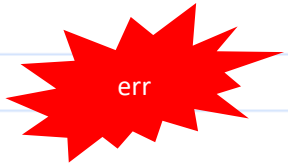
"Enjoy your Coffee"
is shown quickly immediately ...

boilerWater() takes longer than **brewCoffee**()

brewCoffee(water) executes without waiting "Hot water"

Not what we want

```
> function boilWater() {
      console.log("Boiling water...");
      setTimeout(() => {
          console.log("Hot water")
          return "Hot water";
      }, 5000);
    }

  function brewCoffee(water) {
      console.log(`Brewing coffee with ${water}`);
      setTimeout(() => {
          console.log("Coffee")
          return "Coffee";
      }, 2000);

  }

  function makeCoffee() {
      const water = boilWater();
      const coffee = brewCoffee(water);
      console.log(`Enjoy your ${coffee}`);
  }

  makeCoffee();
  Boiling water...                                  VM20914:2
  Brewing coffee with undefined                     VM20914:10
  Enjoy your undefined                              VM20914:21
<· undefined
  Coffee                                            VM20914:12
  Hot water                                         VM20914:4
>
```



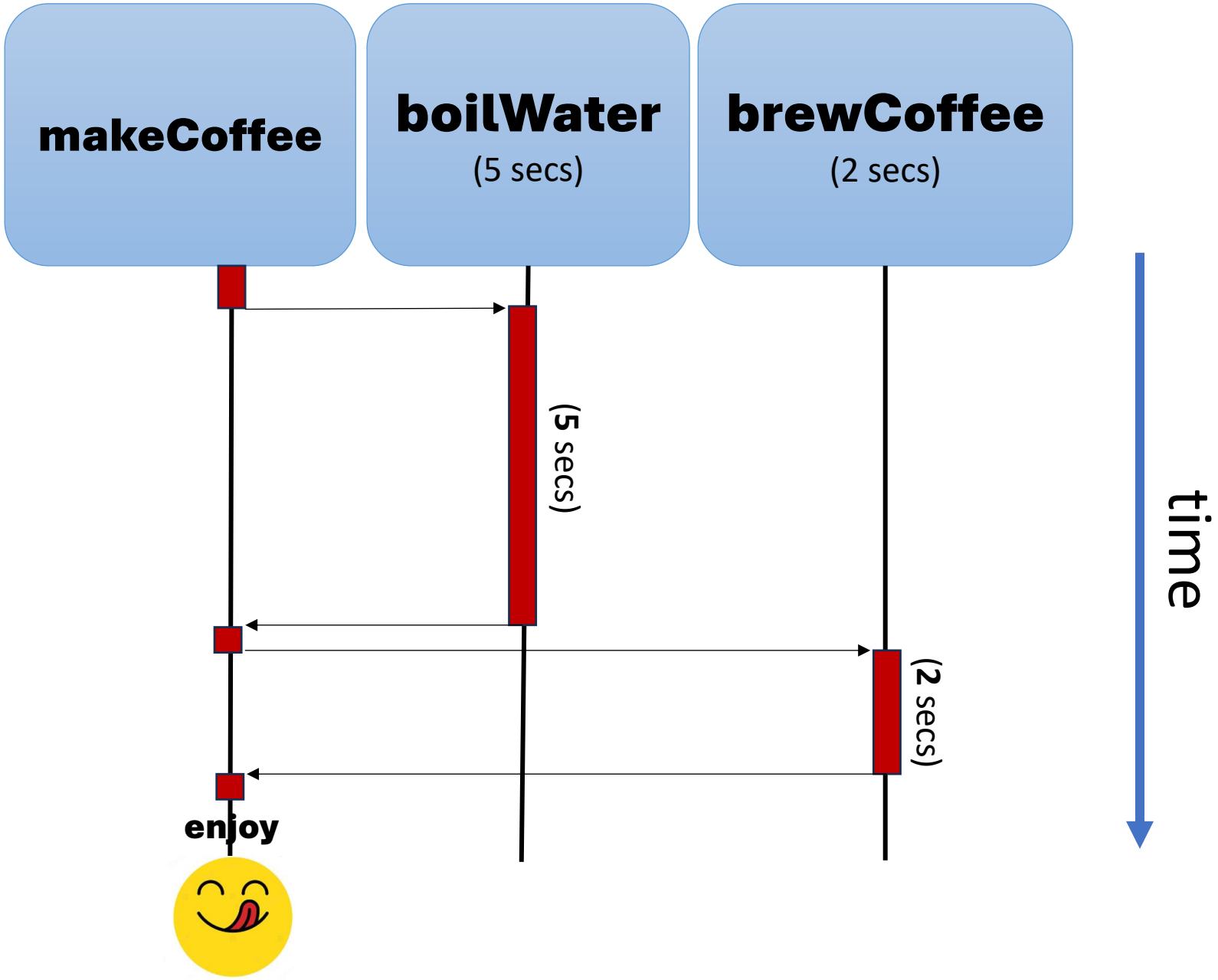
Why is `water` not defined ?

Same with `coffee`?

Solution
`async` / `await`

Let’s make **brewCoffee**() waits to finish executing **boilWater**().

Let’s make **brewCoffee**() waits to finish executing **boilWater**().



Let’s make **brewCoffee**() waits to finish executing **boilWater**() by using `async` / `await` :

```
async function makeCoffee() {
    console.log("Starting coffee...");
    const water = await boilWater();
    const coffee = await brewCoffee(water);
    console.log(`Enjoy your ${coffee}`);
}
```

Some statements are inherently **promises** :

```
fetch("https://fakestoreapi.com/products")
  .then(response => response.json())
  .then(data => console.log(data));
```

With the help of `promise` we can handle asynchronous processing in regular functions:

```
async function boilWater() {
    console.log("Boiling water...");
    return new Promise((resolve) =>
      setTimeout(() => {
        console.log("Hot water");
        resolve("Hot water");;
      }, 5000)
    );
}
```

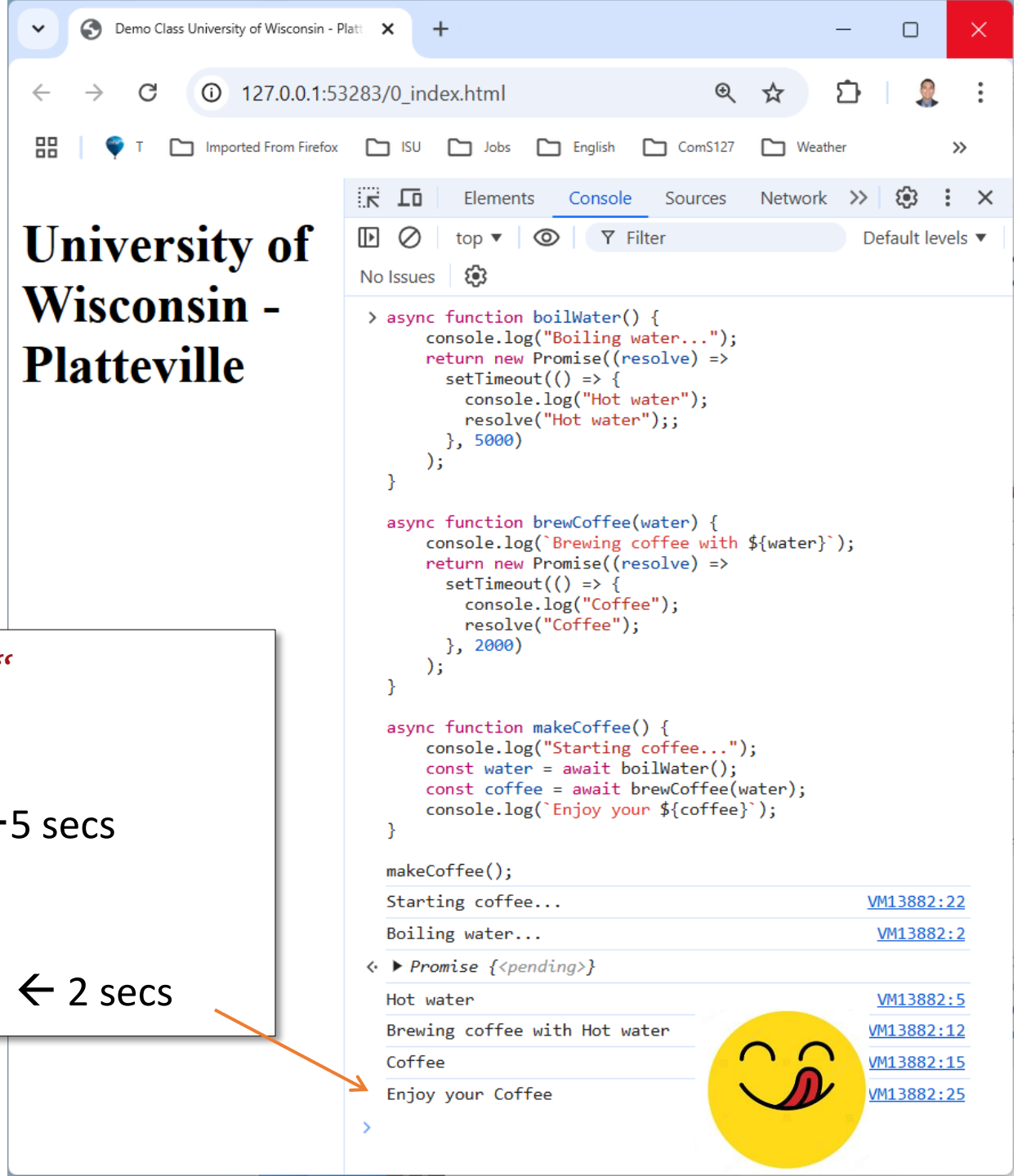
Promise()

setTimeout()

Complete solution using Promise-based setTimeout and `async/await`:

```javascript
async function boilWater() {
    console.log("Boiling water...");
    return new Promise((resolve) =>
      setTimeout(() => {
        console.log("Hot water");
        resolve("Hot water");;
      }, 5000)
    );
}

async function brewCoffee(water) {
    console.log(`Brewing coffee with ${water}`);
    return new Promise((resolve) =>
      setTimeout(() => {
        console.log("Coffee");
        resolve("Coffee");
      }, 2000)
    );
}

async function makeCoffee() {
    console.log("Starting coffee...");
    const water = await boilWater();
    const coffee = await brewCoffee(water);
    console.log(`Enjoy your ${coffee}`);
}

makeCoffee();
```

Try code in a browser:



"Enjoy your Coffee"
is shown after
boilWater() ←5 secs
and
brewCoffee() ← 2 secs

Asynchronous processing

fetch

So far :

- We have use `setTimeout` to simulate processing delay.
- Then, we add `Promise` to use `async`/`await` and have control over asynchronous processing.

Fetch :

- It is a built-in function that allows you to make **network requests** to retrieve data from a server or API.
- It is considered a **promise-based** function.

Demonstrate error in asynchronous execution:

```
function fetchData(){
    fetch("./data.json")
    .then(response=>response.json())
    .then(data => console.log(data));

    for (const person of data)
        console.log(person.firstName);
}

fetchData();
```

{JSON}
JavaScript Object Notation

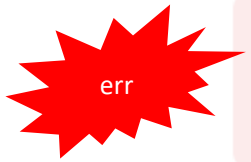
```
[
    {
        "firstName":"Abraham",
        "lastName" : "Aldaco"
    },
    {
        "firstName":"John",
        "lastName" : "Doe"
    },
    {
        "firstName":"Clark",
        "lastName" : "Kent"
    }
]
```

err

```
Uncaught ReferenceError: data is not defined        index.html:18
    at fetchData (index.html:18:34)
    at index.html:32:9
```

```
function fetchData(){
    fetch("./data.json")
    .then(response=>response.json())
    .then(data => console.log(data));

    for (const person of data)
        console.log(person.firstName);
}

fetchData();
```

Why is data not defined ?




```
▶ Uncaught ReferenceError: data is not defined        index.html:18
    at fetchData (index.html:18:34)
    at index.html:32:9
```

Explanation of the error :

execution

```
function fetchData(){
    fetch("./data.json")
    .then(response=>response.json())
    .then(data => console.log(data));

    for (const person of data)
        console.log(person.firstName);
}

fetchData();
```

`fetch` and `console.log` work fine.

But `for` statement is executed **without waiting** for the `fetch` to finish.

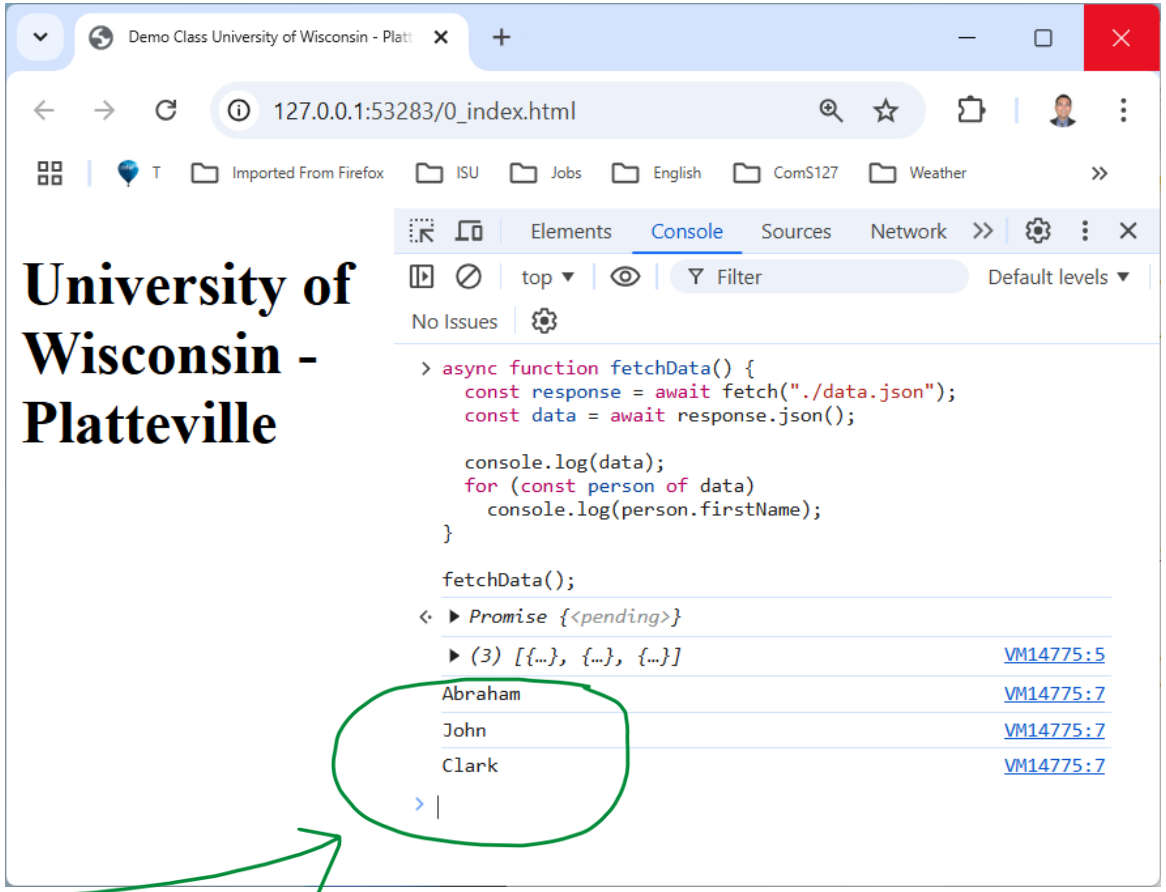
err

```
Uncaught ReferenceError: data is not defined        index.html:18
    at fetchData (index.html:18:34)
    at index.html:32:9
```

Solving the issue adding `async/await`:

execution

```
async function fetchData(){
    const response = await fetch("./data.json");
    const data = await response.json();
    console.log(data);

    for (const person of data)
        console.log(person.firstName);
}

fetchData();
```



Assignment

Convert to `async`/`await` the next code using Promise and setTimeout:

Assignment

```
Area of the room: 150 square units
Total flooring cost: $750
```

```javascript
// Function to calculate the area of a rectangle
function calculateArea(length, width) {
    return length * width;
}

// Function to calculate the cost of flooring based on area
function calculateFlooringCost(area, costPerSquareUnit) {
    return area * costPerSquareUnit;
}

function flooringCost() {

    const length = 10;            // Length of the room
    const width = 15;             // Width of the room
    const costPerSquareUnit = 5; // Cost per square unit

    // Step 1: Calculate the area
    const area = calculateArea(length, width);
    console.log(`Area of the room: ${area} square units`);

    // Step 2: Calculate the flooring cost based on the area
    const totalCost = calculateFlooringCost(area, costPerSquareUnit);
    console.log(`Total flooring cost: $${totalCost}`);
}

flooringCost();
```

Thanks !

Questions ?

Backup

Code

https://github.com/aaldacog/uwp

Promise

A Promise is an object representing the eventual completion or failure of an Asynchronous operation.

Reference:
https://www.mitrais.com/news-updates/asynchronous-in-javascript/
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

Promise

A promise is an object that may produce a single value some time in the future:

either a resolved value, or a reason that it's not resolved (e.g., a network error occurred).

if the promise succeed

if the promise fails

Promise

instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.

1 Let's to execute a **resolve** promise :

```
new Promise(function(resolve, reject) {
    // the function is executed automatically when the promise is constructed
    // after 1 second signal that the job is done with the result "done“
     setTimeout(() => resolve("done"), 1000);
});
```

The Promise executed successfully.
And the result **value** is 'done'

Reference:
https://javascript.info/promise-basics