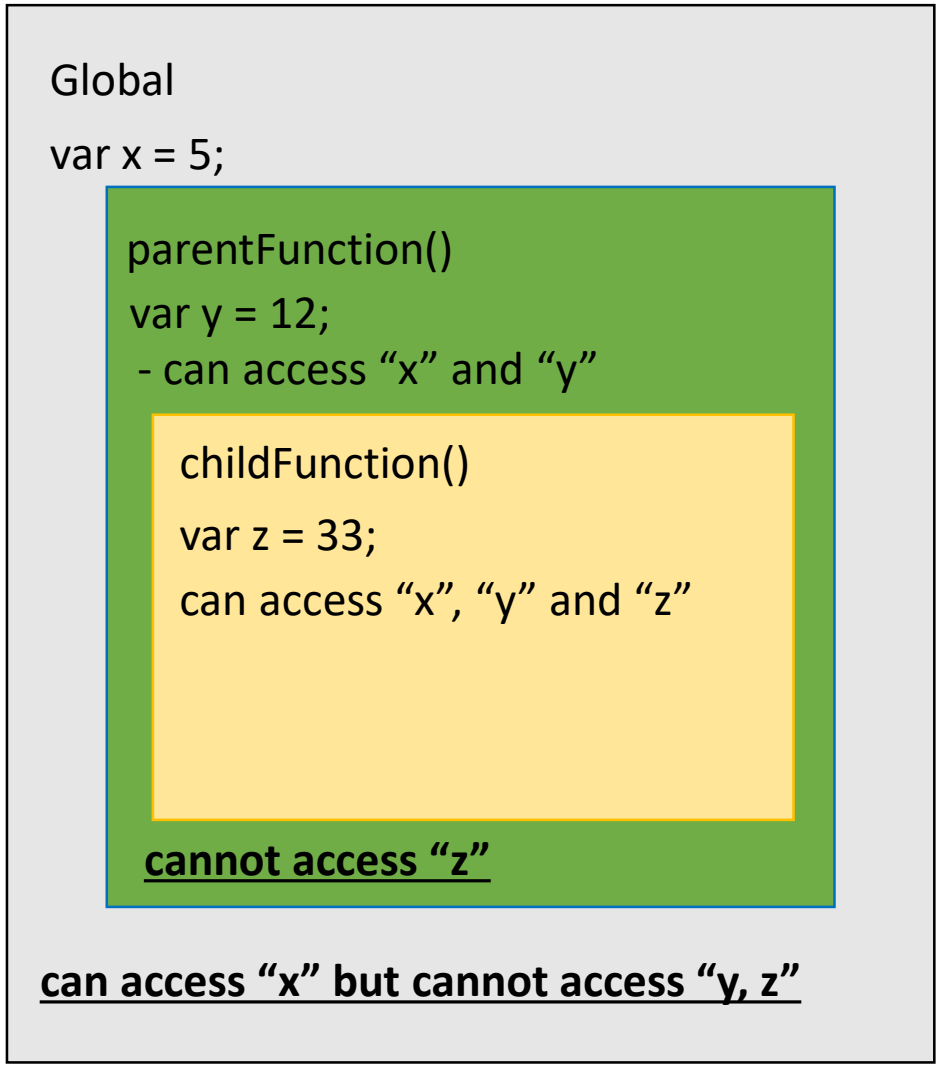
Javascript Essentials for ReactJS

Why we need this?

- Difference between VanillaJS vs React codes
- Understand advance VanillaJS codes
- Create a well-architected code

Scope

- context in which values and expressions are "visible" or can be referenced
- child scopes have access to parent scopes, but not vice versa
- function serves as a closure in JavaScript, and thus creates a scope



Hoisting & Block Scope

- **Hoisting** – the variable and function declarations are put into memory during the compile phase, but stay exactly where you typed them in your code.

```
console.log(hoistingVariable); // returns undefined
var hoistingVariable = 5;
```

- **Block scoped** - is the area within if, switch conditions or for and while loops

```
if (true) {

}

switch (true) {
    default:
        break;
}

for(var counter=0; counter<5; counter++) {

}

while(true) {

}
```

VAR, LET and CONST

	VAR	LET	CONST
Redeclare variable	Yes	No	No
Hoisting	Yes	No	No
Block scope	No	Yes	Yes
Modify Value	Yes	Yes	No

Let’s play with it!

Logical OR “||” & AND “&&”

- Logical OR returns true if 1 of the conditions is true
- Logical AND returns true if all the conditions are true
- But for **javascript** it returns the values that you have used in the condition **(e.g. 5 || 6 === 5)**
- Truthy Values – true, {}, [], “0”, “false”, new Date(), Infinity, -Infinity, non zero numbers
- Falsy Values - false, 0, -0, 0n, “”, null, undefined, NaN

Let’s play with it!

Primitive Values vs Reference Type

- Primitive Values – string, number, bigint, boolean, undefined, and symbol

```
//Primitive Values

let firstNumber = 1;
let copyFirstNumber = firstNumber;
firstNumber = 2;

console.log(firstNumber); //returns 2
console.log(copyFirstNumber); //returns 1
```

- Reference Type – Object and Arrays

```
//Reference Type

let firstArray = [1];
let copyFirstArray = firstArray;
copyFirstArray.push(2)

console.log(firstArray); //returns [ 1, 2 ]
console.log(copyFirstArray); //returns [ 1, 2 ]
```

Let’s play with it!

How can we copy objects/arrays?

- Shallow Copy – we only copy the first layer of the object from the original

- Deep Copy –we have copied the object fully and it does not reference any values in the original

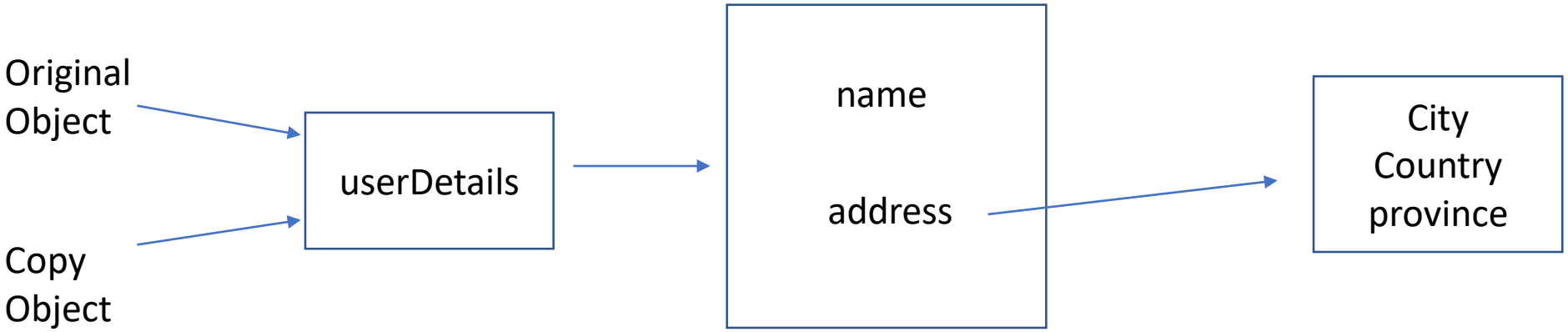
Shallow copy

- To do a shallow copy, we can use the following.
    - Object – Object.assign()
    - Array – Array.slice()
    - Both – spread operator (…)

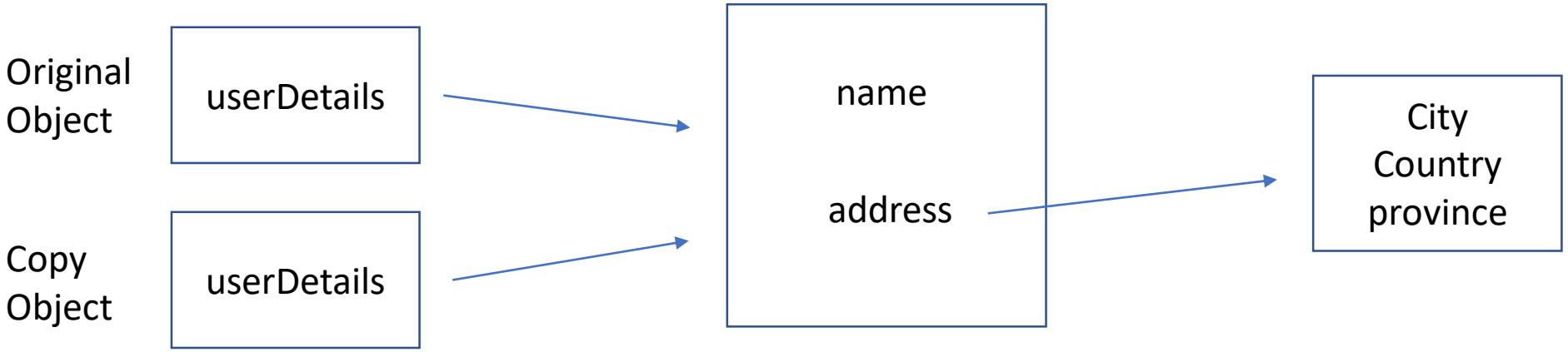
How it works?

```
const object = {
    userDetails: {
        name: 'Firstname Lastname',
        address: {
            city: 'malolos',
            province: 'bulacan',
            country: 'PH'
        }
    }
}
```

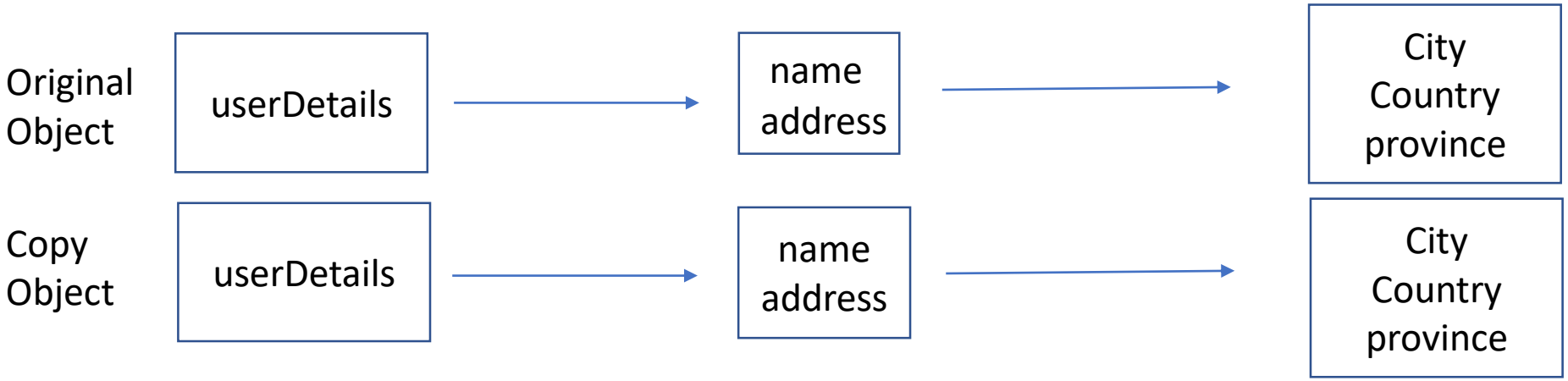
Copy Object = Original Object



Shallow Copy – Object.assign



Shallow Copy and update userDetails (userDetails = { name, address})



Deep copy

As the name states it will do a deep copy of the object so that if you change a property of an object, it will not affect the other copy.

- **JSON.parse(JSON.stringify(object))**
  - bit **slow** in terms of **performance** but the **easiest to code**.
  - will cause some **data loss** if your object contains **function, new Date(), undefined, etc**.
- Packages/Libraries
  - **Lodash** – cloneDeep function
  - **Jquery** – jQuery.extend
- Or CREATE YOUR OWN!

Let’s play with it!

Spread operator

- easily create a **shallow copy** of an object

- mostly use if you want to copy an **object/array** and **add/modify** values in it.

Object

```
const userDetails = {
    name: 'My Name',
    address: {
        city: 'malolos',
        country: 'PH',
    }
}

const newUserDetails = { ...userDetails, newKey: 'newKey' }
```

Array

```
const fruits = [ 'apple', 'mango', 'pineapple' ]

const newFruits = [ ...fruits, 'coconut' ];
```

Rest parameters

- allows us to represent an indefinite number of arguments as **an array**.

```
function sum(...theArgs) {
    return theArgs.reduce((previous, current) => {
        return previous + current;
    });
}

console.log(sum(1, 2, 3)); // 6
```

- Compared with “arguments” object, **rest parameters** can **use array functions** while **arguments cannot**

```
function useArguments(a, b) {
    console.log(arguments[0]); //returns 1
}

useArguments(1, 5);
```

Destructuring

- JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables

- You can also change the name, assign default values and use rest parameters

Without Destructuring

```
const name = userDetails.name;
const address = userDetails.address;
```

```
const firstFruit = fruits[0] || 'defaultFruit';
const otherFruits = fruits.filter((value, index) => index !== 0);
```

With Destructuring

```
const userDetails = {
    name: 'My Name',
    address: {
        city: 'malolos',
        country: 'PH',
    }
}

const { name, address } = userDetails;
```

```
const fruits = [ 'apple', 'mango', 'pineapple'];

const [ firstFruit = 'defaultFruit', ...otherFruits  ] = fruits;
```

Let’s play with it!

Classes

- template for creating objects.

- encapsulate data with code to work on that data.

What do we have in Classes?

- **Constructor** – executed when you initialize the class
- **Functions** – methods you can use using that class
- **Public variables** – can be access when you instantiate the class
- **Private variables** – can only be access inside the class
- **Static variables** – can be called without the need to instantiate the class
- **“this” object** – reference the class itself
- **Inheritance** – use “extends” to get the attributes of a parent class
- **Super** – calling base class attributes

Classes Example

```
class Shape {
    #privateVariable = 'privateVariable'
    publicVariable = 'publicVariable'
    static staticVariable = 'staticVariable'
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    getArea() {
        return 'not implemented';
    }

    getPrivateVariable() {
        return this.#privateVariable;
    }
};

class Square extends Shape {
    getArea() {
        return this.height * this.width;
    }

    getParentArea() {
        return super.getArea();
    }
}

const houseAndLot = new Square(120,100);

console.log(houseAndLot.getParentArea());
console.log(houseAndLot.getArea());
console.log(houseAndLot.getPrivateVariable());
```

Let’s play with it!

Who is “this” object?

- In most cases, the value of this is determined by how a function is called
- In simple terms this object refers to the parent object where you call the function unless you use “binding”

```
this.console.log('I am Global Object') //Global Object

const userDetails = {
    name: 'My Name',
    getName: function() {
        return this.name; // userDetails Object
    }
};

console.log(userDetails.getName());
```

Arrow functions

- is a **compact alternative** to a traditional function expression, but is limited and **can't be used in all situations**.

- Limitations
  - Does not have its own bindings to **this** or **super**.
  - Does not have **arguments object**.
  - **Not** suitable for **call, apply and bind methods**
  - Can not be used as constructors.
  - Can not use yield, within its body.
  - do not default this to the window scope, rather they **execute in the scope they are created**

```
class Person {
    name="My Name"
    printName() {
        console.log('Person:', this.name);
    }

    printNameArrowFn() {
        const userDetails = {
            name: 'user details',
            printName: () => {
                console.log('User Details:', this.name);
            }
        };

        userDetails.printName();
    }
}

const iam = new Person();
iam.printName(); // Person: My Name
iam.printNameArrowFn(); // User Details: My Name
```

Simple terms

- Standard Function
  - “this” object refers to the parent object of the function where it has been executed

```
this.console.log('I am Global Object') //Global Object

const userDetails = {
    name: 'My Name',
    getName: function() {
        return this.name; // userDetails Object
    }
};

console.log(userDetails.getName());
```

- Arrow Function
  - it retains the “this” where it was created

```
const userDetails = {
    name: 'My Name',
    getName: function() {
        const arrowFunction = () => {
            return this.name; // this refers to userDetails
        }

        return arrowFunction();
    }
}

console.log(userDetails.getName());
```

Let’s play with it!

How Javascript works?

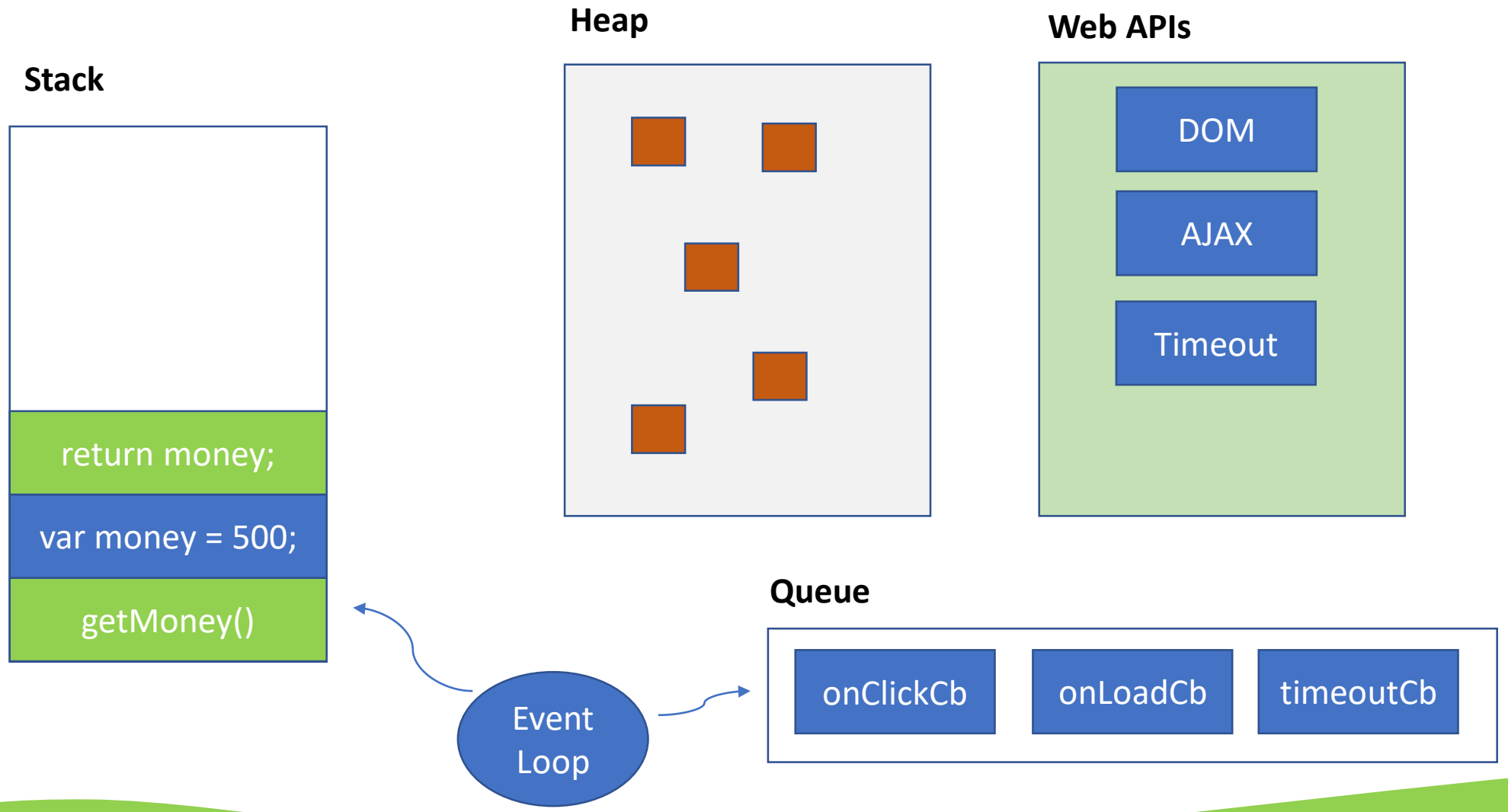
- **Javascript** is **single threaded** and has a **synchronous** execution model

- **Synchronous** – one command/code is being executed at a time

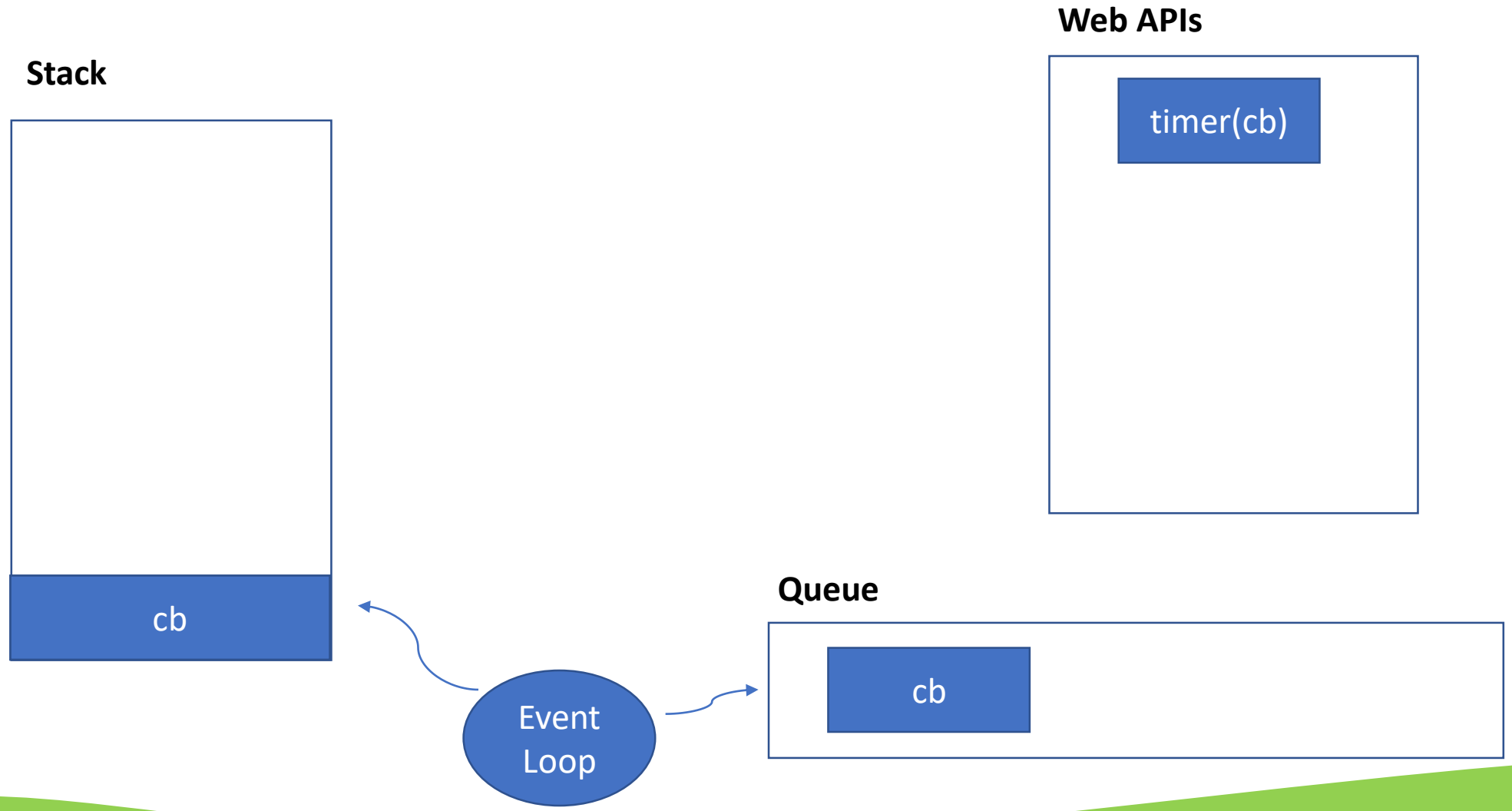
- **Asynchronous** – codes are executed in the background and will give way for others to be executed (e.g. setTimeout)

- **Long running function** will **block** your **code** and your **user** – USE **ASYNCHRONOUS**

Runtime Concept



Example: setTimeout(cb, 5000)



Callback Hell

- **Asynchronous** JavaScript, or JavaScript that uses **callbacks**, is **hard** to get right **intuitively**.

```
asyncFnA(param1, param2, function() {
    asyncFnB(dataFromA, function() {
        asyncFnC(dataFromB, function() {

        });
    });
});
```

Promise

- used to **handle asynchronous invocations** nicely

- Has 3 **different states**
  - **Pending**
  - **Fulfilled**
  - **Rejected**

```
const myPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve('fullfilled');
    }, 500)
});

myPromise.then(data => {
    console.log(data);
}).catch(err => {
    console.log(err);
});
```

Promise API

- **Constructor** – Promise(paramFn) - used to wrap functions that do not support promises
  - **paramFn(resolve, reject) - resolve** if fulfilled, **reject** if rejected

- **Promise.resolve(value)** - returns a new Promise object that is **resolved** with the given **value**

- **Promise.reject(reason)** - returns a new Promise object that is **rejected** with the given **reason**

- **Promise.all(iterable)** – wait for **all** promises to be **resolved**, or for **any** to be **rejected**

- **Promise.allSettled(iterable)** - wait until **all promises have settled** (each may resolve or reject)

- **Promise.any(iterable)** - as soon as **one** of the promises in the iterable **fulfills**, returns a single promise that resolves with the value from that promise

- **Promise.race(iterable)** - wait until **any** of the promises is **resolved or rejected**

Instance Methods

- **then(onFulfilled, onRejected)** – called after the promise is done executing
  - **onFulFilled** – function that will be called when Promise is fulfilled
  - **onRejected** – function that will be called when Promise is rejected

- **catch(reason)** – called when the promise is **rejected** when **onRejected** in **then** is **not present**
  - this will also catch errors when an error occurs in onFulFilled and onRejected

- **finally()** – called **either** promise is **fulfilled or rejected**
  - does not have any parameters. If you want to do something with the **data returned** by **then** or **catch**, then just use **“then” after the catch**

Async/await

- use to make a function asynchronous
- can use “**await**” keyword so you can wait for an asynchronous invocation to finish before executing next codes
- cleaner style than **Promise** as you do not need to do chaining that much
- **returns** a **Promise**

```
async function callerAsync() {
    const dataFromAsync = await ajaxCall();
    const massageData = dataFromAsync + 'my name';
    return massageData;
};
```

Let’s play with it!

Export & Import

- Export functions before declarations

```
export const getSubtotalPrice = () => {

}

export const getBackgroundColorStyleForButton = () => {

}
```

- Export apart from declarations

```
const getSubtotalPrice = () => {

}

const getBackgroundColorStyleForButton = () => {

}

export {
    getSubtotalPrice,
    getBackgroundColorStyleForButton
}
```

- To import them, use the code below

```
import { getSubtotalPrice, getBackgroundColorStyleForButton } from './product';
```

- Export “as”

```
const getSubtotalPrice = () => {

}

const getBackgroundColorStyleForButton = () => {

}

export {
    getSubtotalPrice as getSubPrice,
    getBackgroundColorStyleForButton as getBGStyle
}
```

- Import “as”

```
import { getSubtotalPrice as getSubPrice, getBackgroundColorStyleForButton as getBGStyle } from './product';
```

```
import * as productUtils from './product';

productUtils.getSubtotalPrice();
productUtils.getBackgroundColorStyleForButton();
```

- Export default

```
const getBackgroundColorStyleForButton = () => {

}


export default getBackgroundColorStyleForButton
```

- Importing default export

```
import getBGStyle from './product';

getBGStyle();
```

```
import * as productUtils from './product';

productUtils.default();
```

Summary

- **Scope** – hierarchy on how we can access variables
- **Hoisting** – declaration of variables are added to memory first before execution
- **Block Scope** - is the area within if, switch conditions or for and while loops (let and const)
- **Logical OR, AND** – returns the value in the conditions, not just true/false
- **Primitive Values** – stores the value
- **Reference Type** – stores the reference to the memory (object/arrays)
- **Shallow Copy** – copies only the first layer of an object/array
- **Deep Copy** – copies all the layer of an object/array
- **Spread** – creates a shallow copy of an object/array
- **Rest Parameters** – put all the parameters in an array, use this instead of arguments obj
- **Destructuring** – get keys/data efficiently from object/array

- **Classes -** template for creating objects and encapsulating data
- **this object** - refers to the parent object where you call the function unless you use “binding”
- **Arrow Functions** – it retains the “this” where it was created
- **Asynchronous code** – executed in the Web API and it’s non blocking
- **Callback Hell** – hellish way of adding callback functions to asynchronous invocations
- **Promise** – handle async codes nicely using chaining (then, catch, finally)
- **Async/await** – handle async codes nicely using “await” keyword, async function returns Promise
- **Export/Import** – to create reusable functions that can be used anywhere in your codebase