Named vs Anonymous Functions

Named Functions

```
function greeting(name) {
  return `Hello ${name}!`;
}
```

Named functions always look in the format below:

```
function functionoName() {
  ...
}
```

Anonymous Functions

Anonymous functions are functions that are not assigned names. They can take a few different forms.

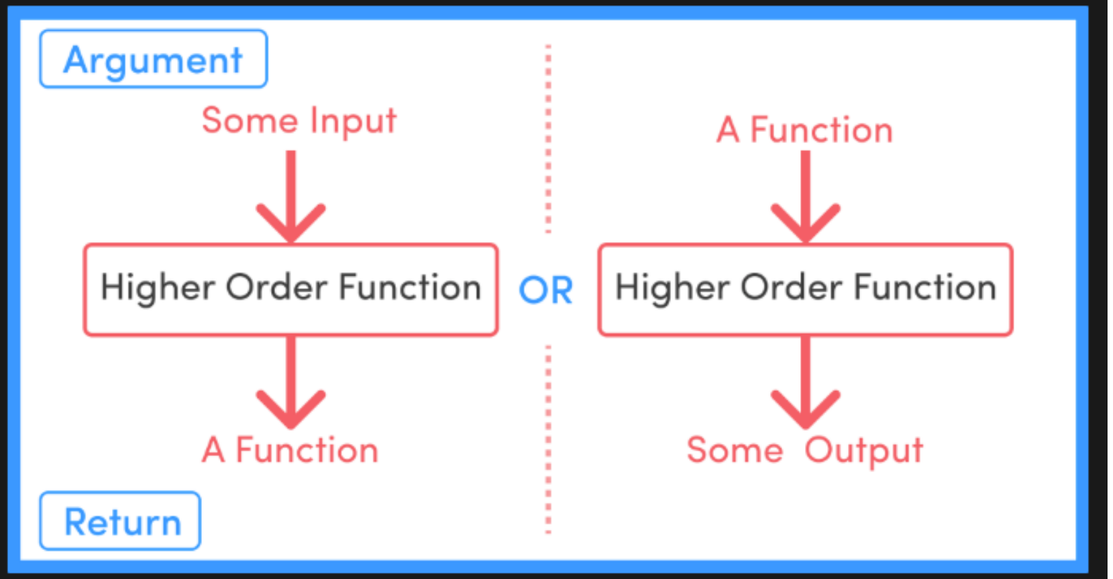
```
const greeting = function (name) {
  return `Hello ${name}!`;
};

const farewell = (name) => {
  return `Bye ${name}!`;
};
```

Although they do not have names, they can be referenced through the variable they are assigned to.

Once initialized, the anonymous function exists through the variable, but the original function itself is gone and cannot be reused.

- **Higher-order functions** are functions that can take other functions as arguments or return functions as their results.



- They enable a functional programming style in JavaScript and are powerful tools for managing and manipulating functions and data.

Example

Example 1: Using map() as a higher-order function

```
let numbers = [1, 2, 3, 4, 5];
let squaredNumbers = numbers.map(function (num) {
  return num * num;
});
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

Example 2: Using filter() as a higher-order function

```
let words = ["apple", "banana", "cherry", "date", "elderberry"];
let filteredWords = words.filter(function (word) {
  return word.length > 5;
});
console.log(filteredWords); // Output: ['banana', 'cherry', 'elderberry']
```

Arrow functions are like a shorter way to write regular functions.

```
function traditionalFunction(x, y) {
  return x + y;
}

const arrowFunction = (x, y) => x + y;

console.log(traditionalFunction(1, 2)); // 3
console.log(arrowFunction(1, 2)); // 3
```

Syntax

```
() => expression

param => expression

(param) => expression

(param1, paramN) => expression

() => {
  statements
}

param => {
  statements
}

(param1, paramN) => {
  statements
}
```

So what are different from traditional functions?

- They don't have their own `this`, `arguments`, or `super`, so they shouldn't be used for certain things.
- You can't make new objects with arrow functions. Trying will cause an error.
- They can't use `yield` or be used to make special kinds of functions.

Let's take a look at a quick example

Here is an example of a `sum` function that returns the sum of two parameters:

```
function sum(a, b) {
  return a + b;
}
```

You can execute the `sum` function before declaring the function due to hoisting:

```
sum(1, 2); // 3

function sum(a, b) {
  return a + b;
}
```

Now let's try with the arrow function

```
sum(1, 2) //Uncaught ReferenceError: Cannot access 'sum' before initialization

const sum = function (a, b) {
  return a + b
}
```

Before we go any further, please take a look at Named vs. Anonymous Functions first.

An *arrow function expression* is an anonymous function expression written with the "fat arrow"

Nested Functions

- Inner functions are functions defined within another function.
- They are also known as nested functions and have access to the outer (enclosing) function's variables and scope.

Example

```
function outerFunction() {
  let outerVar = 10;

  function innerFunction() {
    let innerVar = 5;
    return outerVar + innerVar;
  }

  return innerFunction();
}

let result = outerFunction();
console.log(result); // Output: 15
```

Scopes

- Variables inside a function are only accessible within that function's scope.
- Functions can access variables and functions from their scope and any parent scopes.
- Functions defined in the global scope can access all global variables.
- Functions inside other functions can access variables from their parent function's scope and any other accessible variables.

Scopes

- Variables inside a function are only accessible within that function's scope.
- Functions can access variables and functions from their scope and any parent scopes.
- Functions defined in the global scope can access all global variables.
- Functions inside other functions can access variables from their parent function's scope and any other accessible variables.

```
// The following let variables are defined in the global scope
let mid = 20;
let final = 5;
let fname = 'Ada';

// sum function is defined in the global scope
function sum() {
    return mid + final;
}
console.log(`#1 sum: ${sum()}`); // Returns 25

mid = 10;
console.log(`#2 sum: ${sum()}`); // Returns 15

function getScore() {
    let mid = 10;
    let final = 30;

    // yourScore is a nested function
    function yourScore() {
        return fname + ' scored ' + (mid + final);
    }

    return yourScore;
}

const score = getScore();
console.log(score()); // Returns "Ada scored 40"
```

Closure Functions

- A closure is when a function remembers the environment in which it was created, including the variables outside of its scope.
- It allows an inner function to access and use the variables from its outer function even afte the outer function has finished running.
- Essentially, it's like a way for a function to retain a connection to its birthplace, so it can always access its 'ancestral' (บรรพบุรุษ) data.

Example

```
function createCounter() {
  let count = 0;
  return {
    increment: function() {
      count++;
    },
    decrement: function() {
      count--;
    },
    getCount: function() {
      return count;
    }
  };
}

let counter = createCounter();

counter.increment();
counter.increment();
counter.increment();
console.log(counter.getCount()); // Output: 3

counter.decrement();
console.log(counter.getCount()); // Output: 2
```

forEach()

- The `.forEach()` method calls a function for each element in an array.

Usage

```
array.forEach(function(currentValue, index, array) {
  // Your code here
});
```

Example

Example 1: Printing elements of an array

```
let array = [1, 2, 3, 4, 5];
array.forEach(function(element) {
  console.log(element);
});
```

Example 2: Modifying array elements using forEach

```
let words = ['apple', 'banana', 'cherry'];
words.forEach(function(word, index, arr) {
  arr[index] = word.toUpperCase();
});
console.log(words);
```

map()

- Creates a new array by calling a function for every array element.
- Does not execute the function for empty elements.
- Does not change the original array.

Usage

```
let newArray = array.map(function(currentValue, index, array) {
  // Your code here
  return result; // must return
});
```

Example

Example 1: Doubling each element of an array

```
let numbers = [1, 2, 3, 4, 5];
let doubledNumbers = numbers.map(function (number) {
  return number * 2;
});
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

Example 2: Converting array of strings to uppercase

```
let words = ["apple", "banana", "cherry"];
let uppercaseWords = words.map(function (word) {
  return word.toUpperCase();
});
console.log(uppercaseWords); // Output: ['APPLE', 'BANANA', 'CHERRY']
```

filter()

- Creates a new array with all elements that pass the test implemented by the provided function.

Usage

```
let newArray = array.filter(function(currentValue, index, array) {
  // Your code here
  return condition; // must return
});
```

Example

Example 1: Filtering even numbers from an array

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let evenNumbers = numbers.filter(function (number) {
  return number % 2 === 0;
});
console.log(evenNumbers); // Output: [2, 4, 6, 8, 10]
```

Example 2: Filtering words with more than 5 characters

```
let words = ['apple', 'banana', 'cherry', 'date', 'elderberry'];
let longWords = words.filter(function (word) {
  return word.length > 5;
});
console.log(longWords); // Output: ['banana', 'cherry', 'elderberry']
```

find()

- Returns the value of the first element in an array that satisfies the provided testing fun

Usage

```
let result = array.find(function(currentValue, index, array) {
  // Your code here
  return condition; // must return
});
```

Example

Example 1: Finding the first element that is greater than 4

```
let numbers = [1, 3, 5, 7, 9, 2, 4, 6, 8];
let foundNumber = numbers.find(function (number) {
  return number > 4;
});
console.log(foundNumber); // Output: 5
```

Example 2: Finding a specific string in an array

```
let words = ['apple', 'banana', 'cherry', 'date'];
let foundWord = words.find(function (word) {
  return word === 'cherry';
});
console.log(foundWord); // Output: 'cherry'
```

findIndex()

- Returns the index of the first element in an array that satisfies the provided testing function. If no element passes the test, it returns -1.

Usage

```
let resultIndex = array.findIndex(function(currentValue, index, array) {
  // Your code here
  return condition; // must return
});
```

Example

Example 1: Finding the index of the first element greater than 4

```
let numbers = [1, 3, 5, 7, 9, 2, 4, 6, 8];
let foundIndex = numbers.findIndex(function (number) {
  return number > 4;
});
console.log(foundIndex); // Output: 2
```

Example 2: Finding the index of a specific string in an array

```
let words = ['apple', 'banana', 'cherry', 'date'];
let foundIndexWord = words.findIndex(function (word) {
  return word === 'cherry';
});
console.log(foundIndexWord); // Output: 2
```

`.every()`

- Checks if all elements in an array pass a specified test, returning true if all elements meet the condition, and false if at least one does not.

Usage

```
let result = array.every(function(currentValue, index, array) {
  // Your code here
  return condition;
});
```

Example

Example 1: Checking if all numbers are greater than 5

```
let numbers = [6, 7, 8, 9, 10];
let allGreaterThanFive = numbers.every(function (number) {
  return number > 5;
});
console.log(allGreaterThanFive); // Output: true
```

Example 2: Checking if all words have more than 3 characters

```
let words = ['apple', 'banana', 'cherry', 'date'];
let allMoreThanThree = words.every(function (word) {
  return word.length > 3;
});
console.log(allMoreThanThree); // Output: true
```

reduce()

- `reduce` is used to apply a function to each element in an array to reduce the array to a single value. It returns the final accumulated result.
- This method doesn't run the function if the array is empty, and it doesn't modify the original array.

Usage

```
let result = array.reduce(function(accumulator, currentValue, index, array) {
  // Your code here
  return updatedAccumulator;
}, initialValue);
```

Example

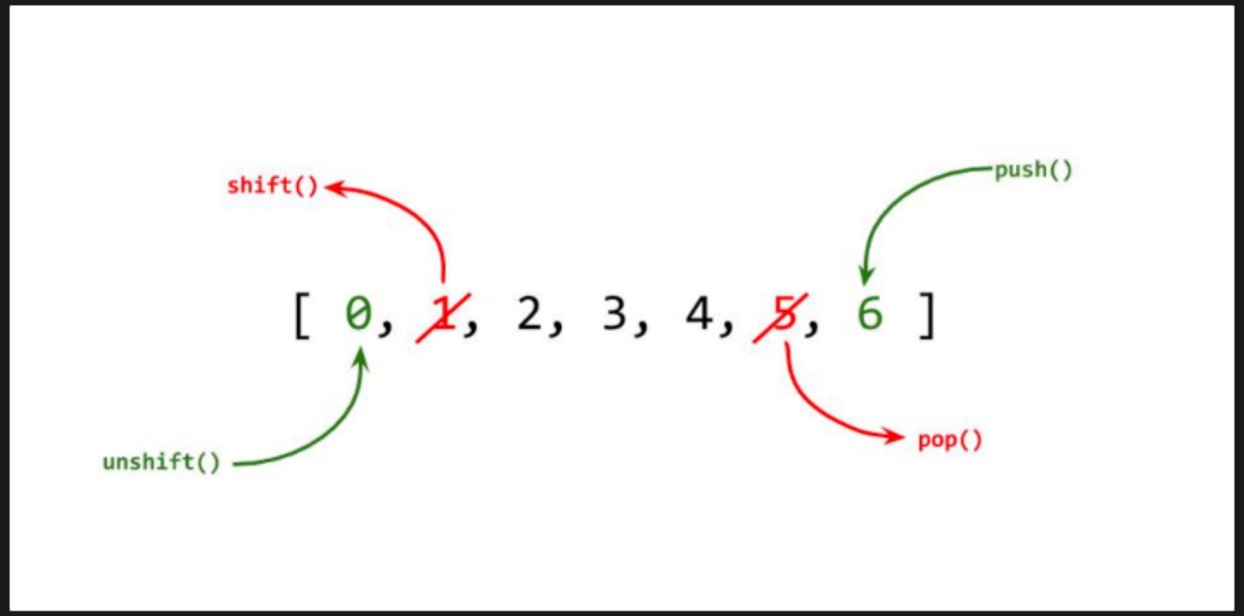
Example 1: Summing all the elements of an array

```
let numbers = [1, 2, 3, 4, 5];
let sum = numbers.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, 0);
console.log(sum); // Output: 15
```

Example 2: Concatenating all elements of an array into a single string

```
let words = ['Hello', ' ', 'world', '!'];
let sentence = words.reduce(function (accumulator, currentValue) {
  return accumulator + currentValue;
}, '');
console.log(sentence); // Output: 'Hello world!'
```

Stack and Queue Methods



.push(), .pop(), .shift() และ .unshift() เพิ่มค่า ตัดค่า ได้ทั้งหน้าและหลังของ Array[...] | by Win Eiwwongcharoen | Medium

pop()

- **Removes the last element** from an array and returns that element. This operation modifies the original array.

Usage

```
let removedElement = array.pop();
```

Example

Example 1: Removing the last element from an array

```
let fruits = ['apple', 'banana', 'cherry'];
let removedFruit = fruits.pop();
console.log(removedFruit); // Output: 'cherry'
console.log(fruits); // Output: ['apple', 'banana']
```

Example 2: Removing and using the last element of an array

```
let numbers = [1, 2, 3, 4, 5];
let lastNumber = numbers.pop();
console.log(lastNumber); // Output: 5
console.log(numbers); // Output: [1, 2, 3, 4]
```

shift()

- **Removes the first element** from an array and returns that removed element. This operation modifies the original array.

Usage

```
let removedElement = array.shift();
```

Example

Example 1: Removing the first element from an array

```
let fruits = ['apple', 'banana', 'cherry'];
let removedFruit = fruits.shift();
console.log(removedFruit); // Output: 'apple'
console.log(fruits); // Output: ['banana', 'cherry']
```

Example 2: Removing and using the first element of an array

```
let numbers = [1, 2, 3, 4, 5];
let firstNumber = numbers.shift();
console.log(firstNumber); // Output: 1
console.log(numbers); // Output: [2, 3, 4, 5]
```

push()

- **Adds one or more elements to the end of an array** and returns the new length of the array.

Usage

```
array.push(element1, element2, ..., elementN);
```

Example

Example 1: Adding elements to an array

```
let fruits = ['apple', 'banana'];
fruits.push('cherry', 'date');
console.log(fruits); // Output: ['apple', 'banana', 'cherry', 'date']
```

Example 2: Adding a single element to an array

```
let numbers = [1, 2, 3];
numbers.push(4);
console.log(numbers); // Output: [1, 2, 3, 4]
```

unshift()

- **Adds one or more elements to the beginning** of an array and returns the new length of the array.

Usage

```
array.unshift(element1, element2, ..., elementN);
```

Example

Example 1: Adding elements to the beginning of an array

```
let fruits = ['apple', 'banana'];
fruits.unshift('cherry', 'date');
console.log(fruits); // Output: ['cherry', 'date', 'apple', 'banana']
```

Example 2: Adding a single element to the beginning of an array

```
let numbers = [3, 4, 5];
numbers.unshift(2);
console.log(numbers); // Output: [2, 3, 4, 5]
```

splice()

- Changes the contents of an array by removing or replacing existing elements and/or adding new elements in place.

Usage

```
array.splice(start, deleteCount, item1, item2, ...);
```

Here,

- `start` : The index at which to start changing the array.
- `deleteCount` : An integer indicating the number of old array elements to remove.
- `item1, item2, ...` : The elements to add to the array.

Example

Example 1: Removing elements from an array

```
let fruits = ['apple', 'banana', 'cherry', 'date'];
let removedItems = fruits.splice(1, 2);
console.log(removedItems); // Output: ['banana', 'cherry']
console.log(fruits); // Output: ['apple', 'date']
```

Example 2: Adding elements to an array

```
let numbers = [1, 2, 3, 7, 8, 9];
numbers.splice(3, 0, 4, 5, 6);
console.log(numbers); // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

slice()

- Extracts a section of an array and returns a new array without modifying the original one.

Usage

```
let newArray = array.slice(start, end);
```

Here,

- `start` : The beginning index at which to begin extraction.
- `end` : The ending index before which to end extraction. The extracted portion will go up to, but not including, the element at this index.

Example

Example 1: Slicing elements from an array

```
let fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry'];
let slicedFruits = fruits.slice(1, 4);
console.log(slicedFruits); // Output: ['banana', 'cherry', 'date']
```

Example 2: Slicing elements from the end of an array

```
let numbers = [1, 2, 3, 4, 5];
let slicedNumbers = numbers.slice(-3);
console.log(slicedNumbers); // Output: [3, 4, 5]
```

fill()

- Changes all elements in an array to a static value, from a start index (default 0) to an end index (default `array.length`). It modifies the original array.

Usage

```
array.fill(value, start, end);
```

Here,

- `value`: The value to fill the array with.
- `start`: Optional. The index to start filling the array. Defaults to 0.
- `end`: Optional. The index to stop filling the array (not inclusive). Defaults to `array.length`.

Example

Example 1: Filling an array with a static value

```
let numbers = [1, 2, 3, 4, 5];
numbers.fill(0);
console.log(numbers); // Output: [0, 0, 0, 0, 0]
```

Example 2: Filling a portion of an array with a static value

```
let array = [1, 2, 3, 4, 5];
array.fill(0, 2, 4);
console.log(array); // Output: [1, 2, 0, 0, 5]
```

- Sorts the elements of an array in place and returns the sorted array.
- The default sort order is ascending, built upon converting elements into strings and comparing their sequences of UTF-16 code units.

Usage

```
array.sort(compareFunction);
```

Example

Example 1: Sorting strings in alphabetical order

```
let fruits = ['banana', 'cherry', 'apple', 'date'];
fruits.sort();
console.log(fruits); // Output: ['apple', 'banana', 'cherry', 'date']
```

Example 2: Sorting numbers in ascending order

```
let numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5];
numbers.sort(function (a, b) {
  return a - b;
});
console.log(numbers); // Output: [1, 1, 2, 3, 4, 5, 5, 6, 9]
```

Example 3: Sorting in a case-insensitive manner

```
let myArray = ["Apple", "banana", "Orange", "cherry"];
myArray.sort(function (a, b) {
  return a.toLowerCase().localeCompare(b.toLowerCase());
});
console.log(myArray); // Output: ["Apple", "banana", "cherry", "Orange"]
```

concat()

- Merge two or more arrays, creating a new array without changing the existing arrays.

Usage

```
let newArray = array1.concat(array2, array3, ..., arrayN);
```

Example

Example 1: Concatenating two arrays

```
let array1 = [1, 2, 3];
let array2 = [4, 5, 6];
let mergedArray = array1.concat(array2);
console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]
```

Example 2: Concatenating multiple arrays

```
let array3 = ['a', 'b', 'c'];
let array4 = ['d', 'e', 'f'];
let array5 = ['g', 'h', 'i'];
let multipleArrays = array3.concat(array4, array5);
console.log(multipleArrays); // Output: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i']
```

indexOf()

- Returns the first index at which a given element can be found in the array, or -1 if it is not present.

Usage

```
let index = array.indexOf(element, start);
```

Here,

- `element` : The element to locate in the array.
- `start` : Optional. The index at which to start the search. If omitted, the search starts from index 0.

Example

Example 1: Finding the index of an element in an array

```
let fruits = ['apple', 'banana', 'cherry', 'date'];
let index1 = fruits.indexOf('cherry');
console.log(index1); // Output: 2
```

Example 2: Finding the index of an element with a specified starting point

```
let numbers = [2, 5, 9, 2];
let index2 = numbers.indexOf(2, 2);
console.log(index2); // Output: 3
```

join()

- Creates and returns a new string by concatenating all the elements in an array. It can be customized to include a specified separator between each element.

Usage

```
let newString = array.join(separator);
```

Example

Example 1: Joining elements of an array into a string

```
let elements = ['Fire', 'Air', 'Water'];
let result1 = elements.join();
console.log(result1); // Output: 'Fire,Air,Water'
```

Example 2: Joining elements of an array with a custom separator

```
let elements2 = ['Fire', 'Air', 'Water'];
let result2 = elements2.join(' - ');
console.log(result2); // Output: 'Fire - Air - Water'
```

1. Mutable Array Methods:
   Mutable array methods are methods that directly modify the original array. They are often used when you want to change the content of an array in place. Here are some common mutable array methods:
   - `push()`: Adds one or more elements to the end of an array.
   - `pop()`: Removes the last element from an array.
   - `shift()`: Removes the first element from an array.
   - `unshift()`: Adds one or more elements to the beginning of an array.
   - `splice()`: Adds or removes elements from anywhere in an array.
   - `sort()`: Sorts the elements of an array in place.
   - `reverse()`: Reverses the order of elements in an array.
   - `fill()`: Fills all the elements of an array with a static value.
   - `copyWithin()`: Copies a sequence of elements within the array to another position within the same array.
   - `set()` (for Typed Arrays): Sets the value of an element in a typed array.

2. Immutable Array Methods:
   Immutable array methods do not modify the original array but return a new array with the desired modifications. These methods are useful when you want to keep the original array intact. Here are some common immutable array methods:
   - `concat()`: Combines two or more arrays and returns a new array.
   - `slice()`: Returns a shallow copy of a portion of an array.
   - `map()`: Creates a new array by applying a function to each element of an array.
   - `filter()`: Creates a new array with all elements that pass a test.
   - `reduce()` and `reduceRight()`: Reduce an array to a single value based on a function.
   - `every()`: Tests whether all elements in an array pass a test.
   - `some()`: Tests whether at least one element in an array passes a test.
   - `find()`: Returns the first element in an array that satisfies a provided testing function.
   - `findIndex()`: Returns the index of the first element in an array that satisfies a provided testing function.
   - `includes()`: Checks if an array contains a specific element.
   - `flat()` and `flatMap()`: Creates a new array with all sub-array elements concatenated into it.

`JSON.stringify`

converts a value to the *JSON* notation

```
const person = {
  name: 'Blue',
};
console.log(JSON.stringify(person)); // { "name": "Blue" }
```

Example of benefits from `JSON.stringify`

- Check if the object is empty

```
const myObj = {};
if (JSON.stringify(myObj) === '{}') {
  console.log('The object is empty!');
}
```

The object is empty!

The result of the above code

Alternate way without using `JSON.stringify`

```
const myObj = {};
if (Object.keys(myObj).length === 0) {
  console.log('The object is empty!');
}
```

Spread (...) in object literals

- In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.

```
const obj1 = { foo: 'bar', x: 42 }
const obj2 = { foo: 'baz', y: 13 }

const clonedObj = { ...obj1 }
// Object { foo: "bar", x: 42 }
const clonedWithReplace = { ...obj1, foo: 'abc' }
// Object { foo: "abc", x: 42 }
const mergedObj = { ...obj1, ...obj2 }
// Object { foo: "baz", x: 42, y: 13 }
```

Note that the property value of obj2 will replace the property value of obj1 in the merged object.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals

Object Destructuring

```
const student = {
  id: 1001,
  fullname: 'Somchai Jaidee',
  email: 'somchai@example.com'
}

let id = student.id
let fullname = student.fullname
let email = student.email

console.log(id) //1001
console.log(fullname) //Somchai Jaidee
console.log(email) //Somchai@example.com
```

destructuring →

```
const student = {
  id: 1001,
  fullname: 'Somchai Jaidee',
  email: 'somchai@example.com'
}

let { id, fullname, email } = student

console.log(id) //1001
console.log(fullname) //Somchai Jaidee
console.log(email) //Somchai@example.com
```

Object Destructuring

- The **destructuring assignment** syntax is a JavaScript expression that makes it possible to **unpack values from arrays, or properties from objects, into distinct variables.**

```
let a, b, rest;
[a, b] = [5, 10]

console.log(a)//5
console.log(b)//10

[a, b, ...rest] = [5, 10, 15, 20, 25]
console.log(rest) // [15,20,25]
```

```
({ a, b } = { a: 10, b: 20 });
console.log(a) // 10
console.log(b) // 20

({ a, b, ...rest } = { a: 10, b: 20, c: 30, d: 40 })
console.log(a) // 10
console.log(b) // 20
console.log(rest) // {c: 30, d: 40}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment

Nested Object Destructuring

```
const msg = {
  sender: 'Somsak',
  recipient: 'Pornsuda',
  content: {
    header: 'Reminder our party',
    body: 'let see you in the party'
  }
}

const {content: { header }} = msg

console.log(header)//Reminder our party
```