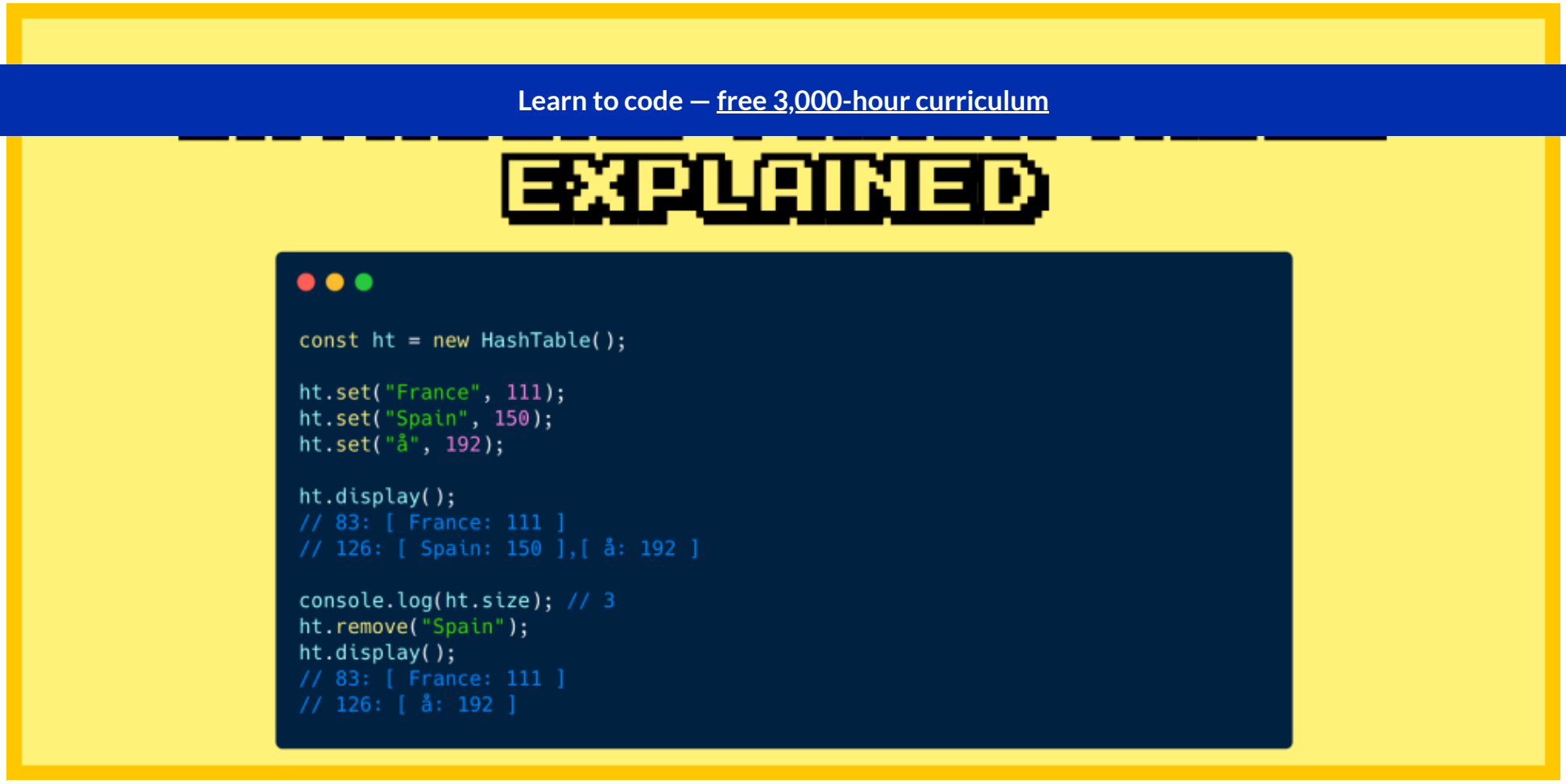
MAY 11, 2021 / #JAVASCRIPT

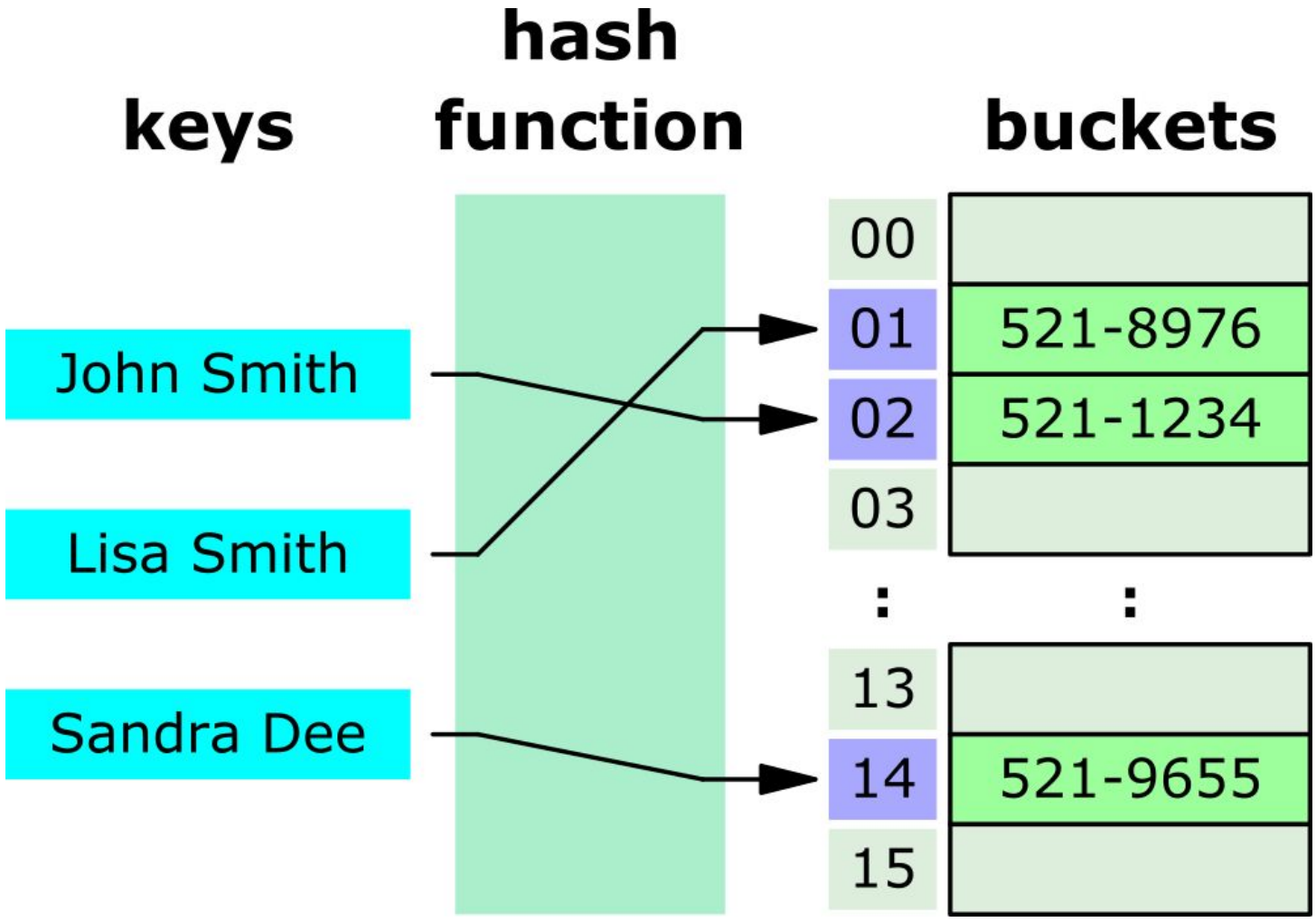
JavaScript Hash Table – Associative Array Hashing in JS

Nathan Sebhastian



Hash Tables are a data structure that allow you to create a list of paired values. You can then retrieve a certain value by using the key for that value, which you put into the table beforehand.

A Hash Table transforms a key into an integer index using a hash function, and the index will decide



Hash table for storing phone books (from Wikipedia)

HASH TABLE TIME COMPLEXITY IN BIG O NOTATION		
Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Source from Wikipedia

This tutorial will help you understand Hash Table implementation in JavaScript as well as how you can build your own Hash Table class.

First, let's look at JavaScript's `Object` and `Map` classes.

How to Use Hash Tables with Object and Map

pair the object's property value with a property key.

In the following example, the key `Nathan` is paired with the phone number value of `"555-0182"` and the key `Jane` is paired with the value `"315-0322"`:

```
let obj = {
  Nathan: "555-0182",
  Jane: "315-0322"
}
```

JavaScript object is an example of Hash Table implementation

But JavaScript's `Object` type is a special kind of Hash Table implementation for two reasons:

- It has properties added by the `Object` class. Keys you input may conflict and overwrite default properties inherited from the class.
- The size of the Hash Table is not tracked. You need to manually count how many properties are defined by the programmer instead of inherited from the prototype.

For example, the `Object` prototype has the `hasOwnProperty()` method which allows you to check


```
const obj = {};
obj.name = "Nathan";

console.log(obj.hasOwnProperty("name")); // true
```

JavaScript object inherited method call example

JavaScript doesn't block an attempt to overwrite the `hasOwnProperty()` method, which may cause an error like this:

```
const obj = {};
obj.name = "Nathan";
obj.hasOwnProperty = true;

console.log(obj.hasOwnProperty("name"));
// Error: obj.hasOwnProperty is not a function
```

JavaScript object inherited property gets overwritten

structure which is called `Map`

Just like `Object`, `Map` allows you to store key-value pairs inside the data structure. Here's an example of `Map` in action:

```
const collection = new Map();

collection.set("Nathan", "555-0182");
collection.set("Jane", "555-0182");

console.log(collection.get("Nathan")); // 555-0182
console.log(collection.size); // 2
```

JavaScript Map class is another implementation of Hash Table

Unlike the `Object` type, `Map` requires you to use the `set()` and `get()` methods to define and retrieve any key-pair values that you want to be added to the data structure.

You also can't overwrite `Map` inherited properties. For example, the following code tried to

overwrite the `size` property value to `false`:


```
const collection = new Map();

collection.set("Nathan", "555-0182");
collection["size"] = false;

console.log(collection.get("size")); // undefined
console.log(collection.size); // 1
```

Map type property can't be overwritten

As you can see from the code above, you can't add a new entry to the `Map` object without using the `set()` method.

The `Map` data structure is also iterable, which means you can loop over the data as follows:

```
const myMap = new Map();

myMap.set("Nathan", "555-0182");
myMap.set("Jane", "315-0322");
```

```
for (let [key, value] of myMap) {
  console.log(`${key} = ${value}`);
```


Iterating through a Map object

Now that you've learned how JavaScript implements Hash Tables in the form of `Object` and `Map` data structures, let's see how you can create your own Hash Table implementation next.

How to Implement a Hash Table Data Structure in JavaScript

Although JavaScript already has two Hash Table implementations, writing your own Hash Table implementation is one of the most common JavaScript interview questions.

You can implement a Hash Table in JavaScript in three steps:

- Create a `HashTable` class with `table` and `size` initial properties
- Add a `hash()` function to transform keys into indices
- Add the `set()` and `get()` methods for adding and retrieving key/value pairs from the table.

Alright, let's start with creating the `HashTable` class. The code below will create a `table` of buckets


```
class HashTable {
  constructor() {
    this.table = new Array(127);
    this.size = 0;
  }
}
```

HashTable class initial properties

All your key/value pairs will be stored inside the `table` property.

How to write the hash() method

Next, you need to create the `hash()` method that will accept a `key` value and transform it into an index.

A simple way to create the hash would be to sum the ASCII code of the characters in the key using the `charCodeAt()` method as follows. Note that the method is named using `_` to indicate that it's a private class:

```
  let hash = 0;
  for (let i = 0; i < key.length; i++) {
    hash += key.charCodeAt(i);
  }
  return hash;
}
```

But since the `HashTable` class only has 127 buckets, this means that the `_hash()` method must return a number between `0 and 127`.

To ensure that the hash value doesn't exceed the bucket size, you need to use the modulo operator as shown below:

```
_hash(key) {
  let hash = 0;
  for (let i = 0; i < key.length; i++) {
    hash += key.charCodeAt(i);
  }
  return hash % this.table.length;
}
```

Now that you have the `_hash()` method completed, it's time to write the `set()` and `get()` methods.

How to write the set() method

To set the key/value pair in your Hash Table, you need to write a `set()` method that accepts `(key, value)` as its parameters:

- The `set()` method will call the `_hash()` method to get the `index` value.
- The `[key, value]` pair will be assigned to the `table` at the specified `index`
- Then, the `size` property will be incremented by one

```
set(key, value) {
  const index = this._hash(key);
  this.table[index] = [key, value];
  this.size++;
}
```

Now that the `set()` method is complete, let's write the `get()` method to retrieve a value by its key.

How to write the get() method

To get a certain value from the Hash Table, you need to write a `get()` method that accepts a `key` value as its parameter:

- The method will call the `_hash()` method to once again retrieve the table `index`
- Return the value stored at `table[index]`

```
get(key) {
  const index = this._hash(key);
  return this.table[index];
}
```

This way, the `get()` method will return either the key/value pair back or `undefined` when there is no key/value pair stored in the specified `index`.

So far so good. Let's add another method to delete key/value pair from the Hash Table next.

How to write the remove() method

To delete a key/value pair from the Hash Table, you need to write a `remove()` method that accepts

- Retrieve the right `index` using the `_hash()` method
- Check if the `table[index]` has a truthy value and the `length` property is greater than zero. Assign the `undefined` value to the right `index` and decrement the `size` property by one if it is.
- If not, simply return `false`

```
remove(key) {
  const index = this._hash(key);

  if (this.table[index] && this.table[index].length) {
    this.table[index] = undefined;
    this.size--;
    return true;
  } else {
    return false;
  }
}
```

With that, you now have a working `remove()` method. Let's see if the `HashTable` class works

properly.

How to Test the Hash Table Implementation

It's time to test the Hash Table implementation. Here's the full code for the Hash Table implementation again:

```
class HashTable {
  constructor() {
    this.table = new Array(127);
    this.size = 0;
  }

  _hash(key) {
    let hash = 0;
    for (let i = 0; i < key.length; i++) {
      hash += key.charCodeAt(i);
    }
    return hash % this.table.length;
  }

  set(key, value) {
    const index = this._hash(key);
    this.table[index] = [key, value];
    this.size++;
  }
```

```
  get(key) {

  }

  remove(key) {
    const index = this._hash(key);

    if (this.table[index] && this.table[index].length) {
      this.table[index] = [];
      this.size--;
      return true;
    } else {
      return false;
    }
  }
}
```

The HashTable implementation in JavaScript

To test the `HashTable` class, I'm going to create a new instance of the `class` and set some key/value pairs as shown below. The key/value pairs below are just arbitrary number values paired with country names without any special meaning:

```
const ht = new HashTable();
ht.set("Canada", 300);
```


Testing HashTable set() method

Then, let's try to retrieve them using the `get()` method:

```
console.log(ht.get("Canada")); // [ 'Canada', 300 ]
console.log(ht.get("France")); // [ 'France', 100 ]
console.log(ht.get("Spain")); // [ 'Spain', 110 ]
```

Testing HashTable get() method

Finally, let's try to delete one of these values with the `remove()` method:

```
console.log(ht.remove("Spain")); // true
console.log(ht.get("Spain")); // undefined
```

Testing HashTable remove() method

instance and retrieve those values:

```
const ht = new HashTable();

ht.set("Spain", 110);
ht.set("ő", 192);

console.log(ht.get("Spain")); // [ 'ő', 192 ]
console.log(ht.get("ő")); // [ 'ő', 192 ]
```

Hash Table index collision

Oops! Looks like we got into some trouble here. 😳

How to Handle Index Collision

Sometimes, the hash function in a Hash Table may return the same `index` number. In the test case above, the string `"Spain"` and `"ő"` **both return the same** `hash` **value** because the number `507` is

the sum of both of their ASCII code.

Right now, the data stored in our Hash Table implementation looks as follows:

```
[
    [ "Spain", 110],
    [ "France", 100]
]
```

To handle the `index` number collision, you need to store the key/value pair in a second array so that the end result looks as follows:

```
[
    [
        [ "Spain", 110 ],
        [ "å", 192 ]
    ],
    [
        ["France", 100]
```

```
    ],
]
```


To create the second array, you need to update the `set()` method so that it will:

- Look to the `table[index]` and loop over the array values.
- If the key at one of the arrays is equal to the `key` passed to the method, replace the value at index `1` and stop any further execution with the `return` statement.
- If no matching `key` is found, push a new array of key and value to the second array.
- Else, initialize a new array and push the key/value pair to the specified `index`
- Whenever a `push()` method is called, increment the `size` property by one.

The complete `set()` method code will be as follows:

```
set(key, value) {
  const index = this._hash(key);
  if (this.table[index]) {
    for (let i = 0; i < this.table[index].length; i++) {
      // Find the key/value pair in the chain
      if (this.table[index][i][0] === key) {
        this.table[index][i][1] = value;
        return;
```

```
      }
```



```
    this.table[index].push([key, value]);
  } else {
    this.table[index] = [];
    this.table[index].push([key, value]);
  }
  this.size++;
}
```

Next, update the `get()` method so that it will also check the second-level array with a `for` loop and return the right key/value pair:

```
get(key) {
  const target = this._hash(key);
  if (this.table[target]) {
    for (let i = 0; i < this.table.length; i++) {
      if (this.table[target][i][0] === key) {
        return this.table[target][i][1];
      }
    }
  }
  return undefined;
}
```

and remove the array with the right `key` value using the `splice()` method:

```
remove(key) {
  const index = this._hash(key);

  if (this.table[index] && this.table[index].length) {
    for (let i = 0; i < this.table.length; i++) {
      if (this.table[index][i][0] === key) {
        this.table[index].splice(i, 1);
        this.size--;
        return true;
      }
    }
  } else {
    return false;
  }
}
```

With that, your `HashTable` class will be able to avoid any index number collision and store the key/value pair inside the second-level array.

As a bonus, let's add a `display()` method that will display all key/value pairs stored in the Hash Table. You just need to use the `forEach()` method to iterate over the table and `map()` the values to

a string as shown below:


```
display() {
  this.table.forEach((values, index) => {
    const chainedValues = values.map(
      ([key, value]) => `[ ${key}: ${value} ]`
    );
    console.log(`${index}: ${chainedValues}`);
  });
}
```

Here's the complete `HashTable` class code again with the collision avoidance applied for your reference:

```
class HashTable {
  constructor() {
    this.table = new Array(127);
    this.size = 0;
  }

  _hash(key) {
    let hash = 0;
    for (let i = 0; i < key.length; i++) {
      hash += key.charCodeAt(i);
```

```
  }
```



```
  set(key, value) {
    const index = this._hash(key);
    if (this.table[index]) {
      for (let i = 0; i < this.table[index].length; i++) {
        if (this.table[index][i][0] === key) {
          this.table[index][i][1] = value;
          return;
        }
      }
      this.table[index].push([key, value]);
    } else {
      this.table[index] = [];
      this.table[index].push([key, value]);
    }
    this.size++;
  }

  get(key) {
    const index = this._hash(key);
    if (this.table[index]) {
      for (let i = 0; i < this.table.length; i++) {
        if (this.table[index][i][0] === key) {
          return this.table[index][i][1];
        }
      }
    }
    return undefined;
  }
```

```
    if (this.table[index] && this.table[index].length) {
      for (let i = 0; i < this.table.length; i++) {
        if (this.table[index][i][0] === key) {
          this.table[index].splice(i, 1);
          this.size--;
          return true;
        }
      }
    } else {
      return false;
    }
  }

  display() {
    this.table.forEach((values, index) => {
      const chainedValues = values.map(
        ([key, value]) => `[ ${key}: ${value} ]`
      );
      console.log(`${index}: ${chainedValues}`);
    });
  }
}
```

Complete HashTable class implementation

You can test the implementation by creating a new `HashTable` instance and do some insertion and


```
const ht = new HashTable();

ht.set("France", 111);
ht.set("Spain", 150);
ht.set("ǻ", 192);

ht.display();
// 83: [ France: 111 ]
// 126: [ Spain: 150 ],[ ǻ: 192 ]

console.log(ht.size); // 3
ht.remove("Spain");
ht.display();
// 83: [ France: 111 ]
// 126: [ ǻ: 192 ]
```

Another HashTable test

Now there's no collision inside the `HashTable` instance. Great work!

Conclusion

and `Map` data structure.

You've also learned how to implement your own `HashTable` class as well as how to prevent the Hash Table's key indices from colliding by using the chaining technique.

By using a Hash Table data structure, you will be able to create an associative array with fast search, insertion, and delete operations. 😉

Thanks for reading this tutorial

If you want to learn more about JavaScript, you may want to check out my site at sebhastian.com, where I have published over 100 tutorials about programming with JavaScript, all using easy-to-understand explanations and code examples.

The tutorials include String manipulation, Date manipulation, Array and Object methods, JavaScript algorithm solutions, and many more.

Nathan Sebhastian

JavaScript Full Stack Developer currently working with fullstack JS using React and Express. Nathan loves to write about his experience in programming to help other people.

10 to the Power of 0

Recursion

Git Reset to Remote

ISO File

R Value in Statistics

ADB

What is Economics?

MBR VS GPT

Module Exports

Debounce

Python VS JavaScript

Helm Chart

Model View Controller

80-20 Rule

React Testing Library

OSI Model

ASCII Table Chart

HTML Link Code

Data Validation

SDLC

Inductive VS Deductive

JavaScript Keycode List

JavaScript Empty Array

JavaScript Reverse Array

Best Instagram Post Time

How to Screenshot on Mac

Garbage Collection in Java

How to Reverse Image Search

Auto-Numbering in Excel

Ternary Operator JavaScript

Our Nonprofit