Introducing Javascript Tips

> *New year, new project.* ***A JS tip per day!***

With great excitement, I introduce these short and useful daily Javascript tips that will allow you to improve your code writing. With less than 2 minutes each day, you will be able to read about performance, frameworks, conventions, hacks, interview questions and all the items that the future of this awesome language holds for us.

At midday, no matter if it is a weekend or a holiday, a tip will be posted and tweeted.

Can you help us enrich it?

Please feel free to send us a PR with your own Javascript tip to be published here. Any improvements or suggestions are more than welcome! Click to see the instructions

Let's keep in touch

To get updates, watch the repo and follow the Twitter account, only one tweet will be sent per day. It is a deal!

> *Don't forget to Star the repo, as this will help to promote the project!*

Tips list

#06 - Writing a single method for arrays or single elements

> *2016-01-06 by @mattfxyz*

Rather than writing separate methods to handle an array and a single element parameter, write your functions so they can handle both. This is similar to how some of jQuery's functions work (`css` will modify everything matched by the selector).

You just have to concat everything into an array first. `Array.concat` will accept an array or a single element.

```
function printUpperCase(words) {
  var elements = [].concat(words);
  for (var i = 0; i < elements.length; i++) {
    console.log(elements[i].toUpperCase());
  }
}
```

`printUpperCase` is now ready to accept a single node or an array of nodes as it's parameter.

```
printUpperCase("cactus");
// => CACTUS
printUpperCase(["cactus", "bear", "potato"]);
// => CACTUS
```

```
//  BEAR
//  POTATO
```

#05 - Differences between `undefined` and `null`

> *2016-01-05 by @loverajoel*

- `undefined` means a variable has not been declared, or has been declared but has not yet been assigned a value
- `null` is an assignment value that means "no value"
- Javascript sets unassigned variables with a default value of `undefined`
- Javascript never sets a value to `null`. It is used by programmers to indicate that a `var` has no value.
- `undefined` is not valid in JSON while `null` is
- `undefined` typeof is `undefined`
- `null` typeof is an `object`
- Both are primitives
- You can know if a variable is undefined

  ```javascript
  typeof variable === "undefined"
```

```
- You can check if a variable is [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)

  ```javascript
  variable === null
```

- The **equality** operator considers them equal, but the **identity** doesn't

  ```javascript
  null == undefined // true

  null === undefined // false
```

```
## #04 - Sorting strings with accented characters

> 2016-01-04 by [@loverajoel](https://twitter.com/loverajoel)

Javascript has a native method **[sort](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort)** that allows sorting arrays. Doing a simple `array.sort()` will treat each array entry as a string and sort it alphabetically. Also you can provide your [own custom sorting](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort#Parameters) function.
```
```

```javascript
['Shanghai', 'New York', 'Mumbai', 'Buenos Aires'].sort();
// ["Buenos Aires", "Mumbai", "New York", "Shanghai"]
```

But when you try order an array of non ASCII characters like this `['é', 'a', 'ú', 'c']`, you will obtain a strange result `['c', 'e', 'á', 'ú']`. That happens because sort works only with english language.

See the next example:

```javascript
// Spanish
['único','árbol', 'cosas', 'fútbol'].sort();
// ["cosas", "fútbol", "árbol", "único"] // bad order

// German
['Woche', 'wöchentlich', 'wäre', 'Wann'].sort();
// ["Wann", "Woche", "wäre", "wöchentlich"] // bad order
```

Fortunately, there are two ways to overcome this behavior localeCompare and Intl.Collator provided by ECMAScript Internationalization API.

> *Both methods have their own custom parameters in order to configure it to work adequately.*

Using `localeCompare()`

```javascript
['único','árbol', 'cosas', 'fútbol'].sort(function (a, b) {
  return a.localeCompare(b);
});
// ["árbol", "cosas", "fútbol", "único"]

['Woche', 'wöchentlich', 'wäre', 'Wann'].sort(function (a, b) {
  return a.localeCompare(b);
});
// ["Wann", "wäre", "Woche", "wöchentlich"]
```

Using `Intl.Collator()`

```javascript
['único','árbol', 'cosas', 'fútbol'].sort(Intl.Collator().compare);
// ["árbol", "cosas", "fútbol", "único"]

['Woche', 'wöchentlich', 'wäre', 'Wann'].sort(Intl.Collator().compare);
// ["Wann", "wäre", "Woche", "wöchentlich"]
```

- For each method you can customize the location.
- According to Firefox Intl.Collator is faster when comparing large numbers of strings.

So when you are working with arrays of strings in a language other than English, remember to use this method to avoid unexpected sorting.

#03 - Improve Nested Conditionals

> *2016-01-03 by AlbertoFuente*

How can we improve and make more efficient nested `if` statement on javascript.

```
if (color) {
  if (color === 'black') {
    printBlackBackground();
  } else if (color === 'red') {
    printRedBackground();
  } else if (color === 'blue') {
    printBlueBackground();
  } else if (color === 'green') {
    printGreenBackground();
  } else {
    printYellowBackground();
  }
}
```

One way to improve the nested `if` statement would be using the `switch` statement. Although it is less verbose and is more ordered, It's not recommended to use it because it's so difficult to debug errors, here's why.

```
switch(color) {
  case 'black':
    printBlackBackground();
    break;
  case 'red':
    printRedBackground();
    break;
  case 'blue':
    printBlueBackground();
    break;
  case 'green':
    printGreenBackground();
    break;
  default:
    printYellowBackground();
}
```

But what if we have a conditional with several checks in each statement? In this case, if we like to do less verbose and more ordered, we can use the conditional `switch`. If we pass `true` as parameter to the `switch` statement, It allows us to put a conditional in each case.

```
switch(true) {
  case (typeof color === 'string' && color === 'black'):
    printBlackBackground();
    break;
  case (typeof color === 'string' && color === 'red'):
    printRedBackground();
    break;
  case (typeof color === 'string' && color === 'blue'):
```

```
    printBlueBackground();
    break;
  case (typeof color === 'string' && color === 'green'):
    printGreenBackground();
    break;
  case (typeof color === 'string' && color === 'yellow'):
    printYellowBackground();
    break;
}
```

But we must always avoid having several checks in every condition, avoiding use of `switch` as far as possible and take into account that the most efficient way to do this is through an `object`.

```
var colorObj = {
  'black': printBlackBackground,
  'red': printRedBackground,
  'blue': printBlueBackground,
  'green': printGreenBackground,
  'yellow': printYellowBackground
};

if (color && colorObj.hasOwnProperty(color)) {
  colorObj[color]();
}
```

Here you can find more information about this.

#02 - ReactJs - Keys in children components are important

> *2016-01-02 by @loverajoel*

The key is an attribute that you must pass to all components created dynamically from an array. It's unique and constant id that React use for identify each component in the DOM and know that it's a different component and not the same one. Using keys will ensure that the child component is preserved and not recreated and prevent that weird things happens.

> *Key is not really about performance, it's more about identity (which in turn leads to better performance). randomly assigned and changing values are not identity Paul O'Shannessy*

- Use an exisiting unique value of the object.
- Define the keys in the parent components, not in child components

```
//bad
...
render() {
    <div key={{item.key}}>{{item.name}}</div>
}
...
```

```
//good
<MyComponent key={item.key}/>
```

- Use the array index is a bad practice.
- `random()` will not work

```
//bad
<MyComponent key={Math.random()}/>
```

- You can create your own unique id, be sure that the method be fast and attach it to your object.
- When the amount of child are big or involve expensive components, use keys has performance improvements.
- You must provide the key attribute for all children of ReactCSSTransitionGroup.

#1 - AngularJs: `$digest` vs `$apply`

> *2016-01-01 by @loverajoel*

One of the most appreciated features of AngularJs is the two way data binding. In order to make this work AngularJs evaluate the changes between the model and the view through of cycles(`$digest`). You need to understand this concept in order to understand how the framework works under the hood.

Angular evaluate each watcher whenever one event was fired, this is the known `$digest` cycle. Sometimes you have to force to run a new cycle manually and you must choose the correct option because this phase is one of the most influential in terms of performance.

`$apply`

This core method lets you to start the digestion cycle explicitly, that means that all watchers are checked, the entire application starts the `$digest loop` . Internally after execute an optional function parameter, call internally to `$rootScope.$digest();` .

`$digest`

In this case the `$digest` method starts the `$digest` cycle for the current scope and its children. You should notice that the parents scopes will not be checked and not be affected.

Recommendations

- Use `$apply` or `$digest` only when browser DOM events have triggered outside of AngularJS.
- Pass a function expression to `$apply` , this have a error handling mechanism and allow integrate changes in the digest cycle

```
$scope.$apply(() => {
    $scope.tip = 'Javascript Tip';
});
```

- If only needs update the current scope or its children use `$digest`, and prevent a new digest cycle for the whole application. The performance benefit it's self evident
- `$apply()` is hard process for the machine and can lead to performance issues when having a lot of binding.
- If you are using >AngularJS 1.2.X, use `$evalAsync` is a core method that will evaluate the expression during the current cycle or the next. This can improve your application's performance.

#0 - Insert item inside an Array

> *2015-12-29*

Insert an item into an existing array is a daily common task. You can add elements to the end of an array using push, to the beginning using unshift, or the middle using splice.

But those are known methods, doesn't mean there isn't a more performant way, here we go...

Add a element at the end of the array is easy with push(), but there is a way more performant.

```
var arr = [1,2,3,4,5];

arr.push(6);
arr[arr.length] = 6; // 43% faster in Chrome 47.0.2526.106 on macOS 10.11.1
```

Both methods modify the original array. Don't believe me? Check the jsperf

Now we are trying to add an item to the beginning of the array

```
var arr = [1,2,3,4,5];

arr.unshift(0);
[0].concat(arr); // 98% faster in Chrome 47.0.2526.106 on macOS 10.11.1
```

Here is a little bit detail, unshift edit the original array, concat return a new array. jsperf

Add items at the middle of an array is easy with splice and is the most performant way to do it.

```
var items = ['one', 'two', 'three', 'four'];
items.splice(items.length / 2, 0, 'hello');
```

I tried to run these tests in various Browsers and OS and the results were similar. I hope this tips will be useful for you and encourage to perform your own tests!

License