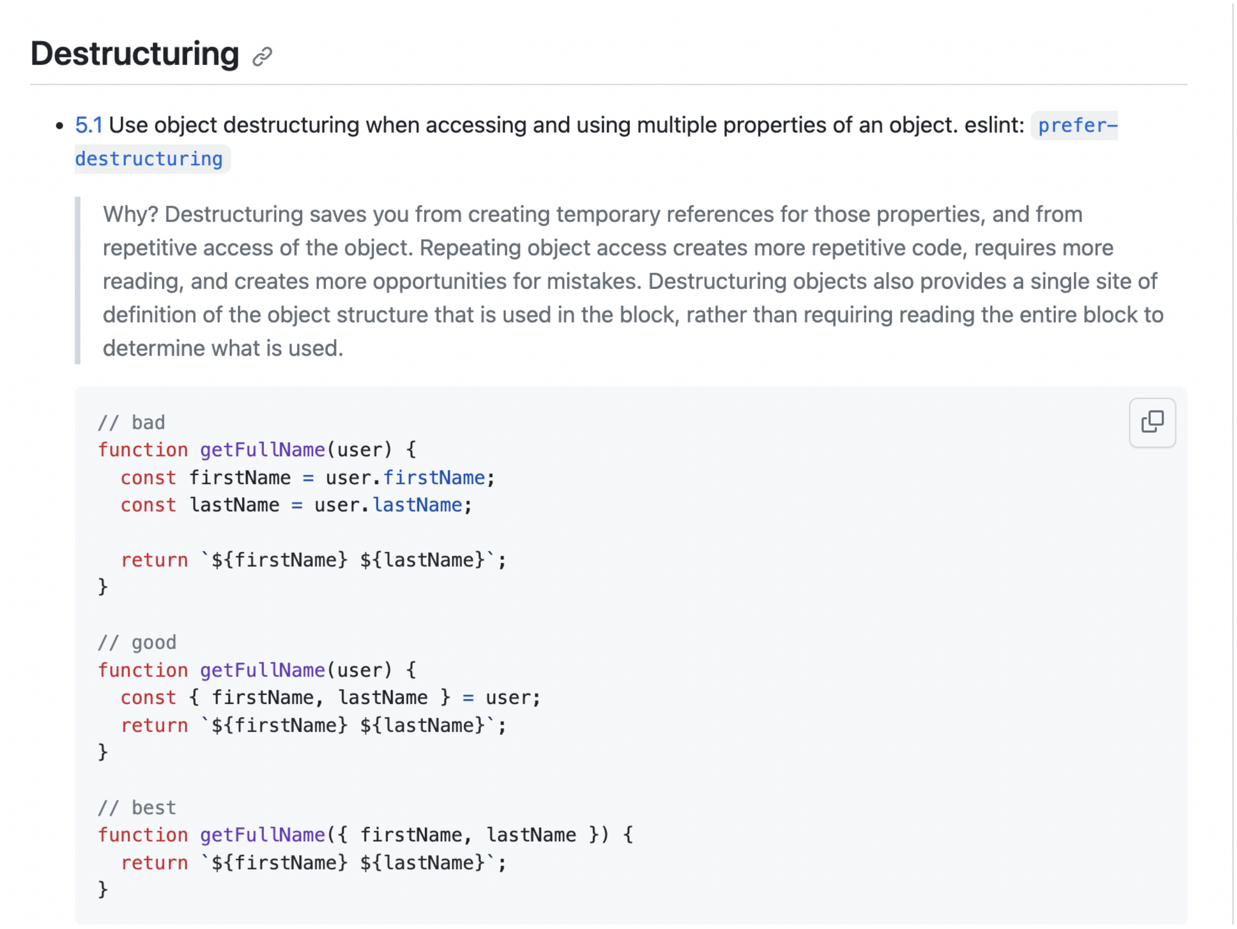
DWA_04.3 Knowledge Check_DWA4

1. Select three rules from the Airbnb Style Guide that you find **useful** and explain why.



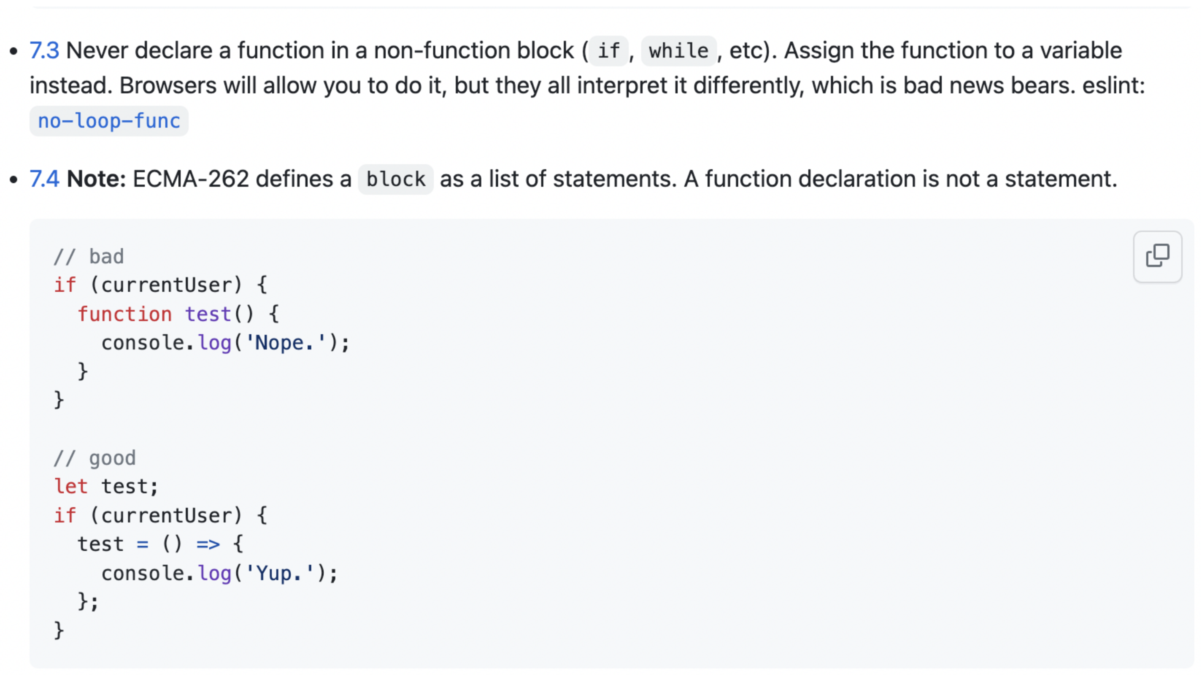
Destructuring

- 5.1 Use object destructuring when accessing and using multiple properties of an object. eslint: `prefer-destructuring`

> Why? Destructuring saves you from creating temporary references for those properties, and from repetitive access of the object. Repeating object access creates more repetitive code, requires more reading, and creates more opportunities for mistakes. Destructuring objects also provides a single site of definition of the object structure that is used in the block, rather than requiring reading the entire block to determine what is used.

```javascript
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

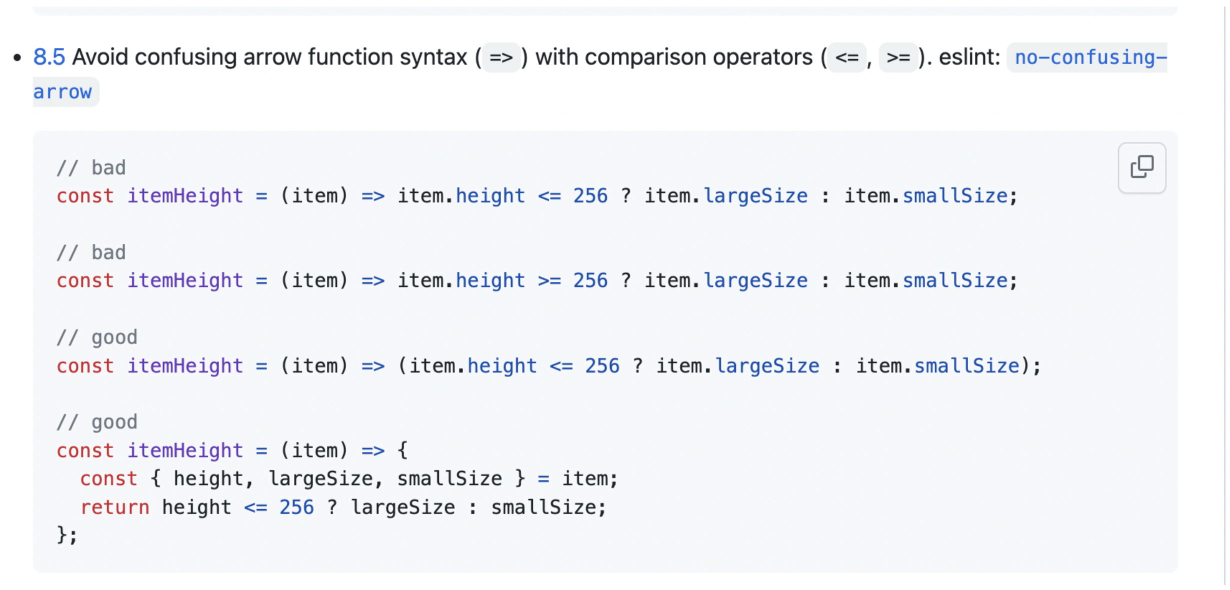
I find this to be a really illuminating example of a complex concept. Thai is also the first time I have seen destructing within a parameter itself.



- 7.3 Never declare a function in a non-function block (`if`, `while`, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears. eslint: `no-loop-func`
- 7.4 **Note:** ECMA-262 defines a `block` as a list of statements. A function declaration is not a statement.

```
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

This stood out to me as something that I've perhaps done in the past and have not understood the consequences of.

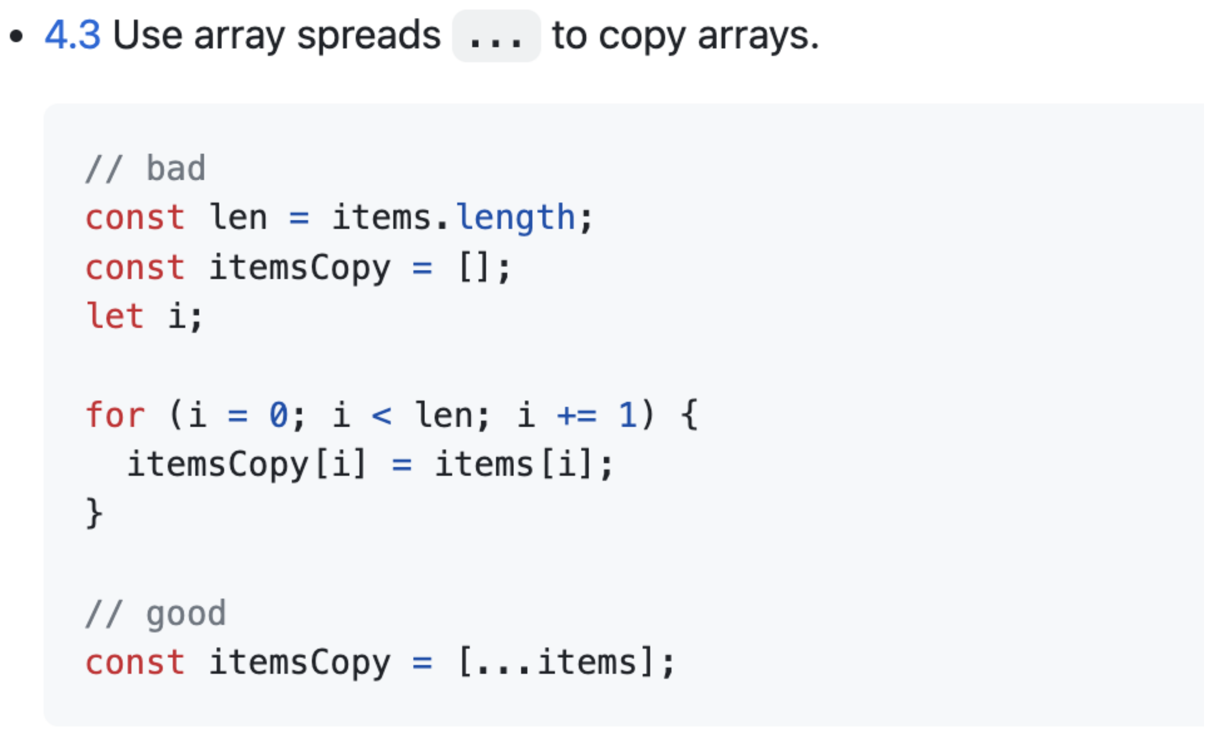


- 8.5 Avoid confusing arrow function syntax (`=>`) with comparison operators (`<=`, `>=`). eslint: `no-confusing-arrow`

```
// bad
const itemHeight = (item) => item.height <= 256 ? item.largeSize : item.smallSize;

// bad
const itemHeight = (item) => item.height >= 256 ? item.largeSize : item.smallSize;

// good
const itemHeight = (item) => (item.height <= 256 ? item.largeSize : item.smallSize);

// good
const itemHeight = (item) => {
  const { height, largeSize, smallSize } = item;
  return height <= 256 ? largeSize : smallSize;
};
```

Seeing this illustrated it is definitely a good approach as the arrow function can be easily confused with comparison operators.

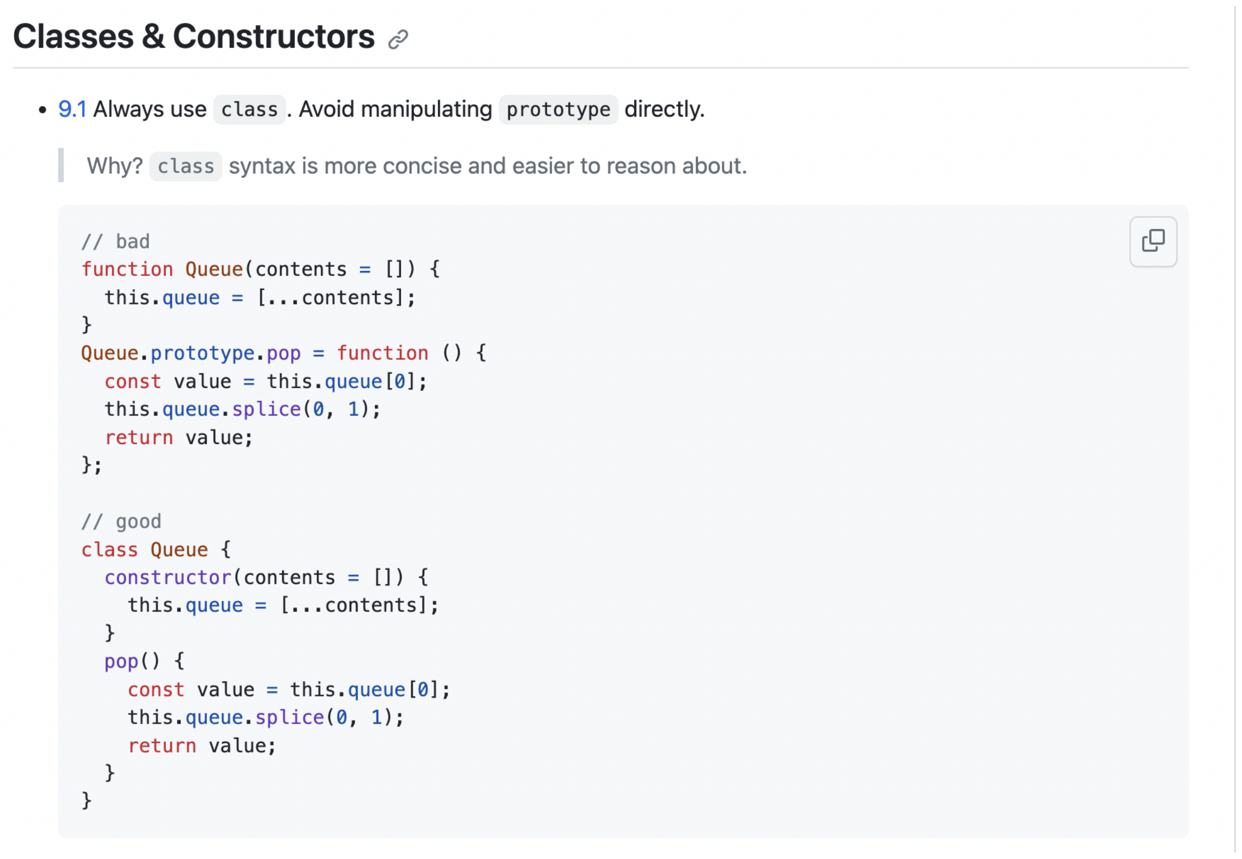
2. Select three rules from the Airbnb Style Guide that you find **confusing** and explain why.



- 4.3 Use array spreads `...` to copy arrays.

```
// bad
const len = items.length;
const itemsCopy = [];
let i;

for (i = 0; i < len; i += 1) {
  itemsCopy[i] = items[i];
}

// good
const itemsCopy = [...items];
```

I’m not sure I entirely understand how the ellipsis is used within JavaScript yet.



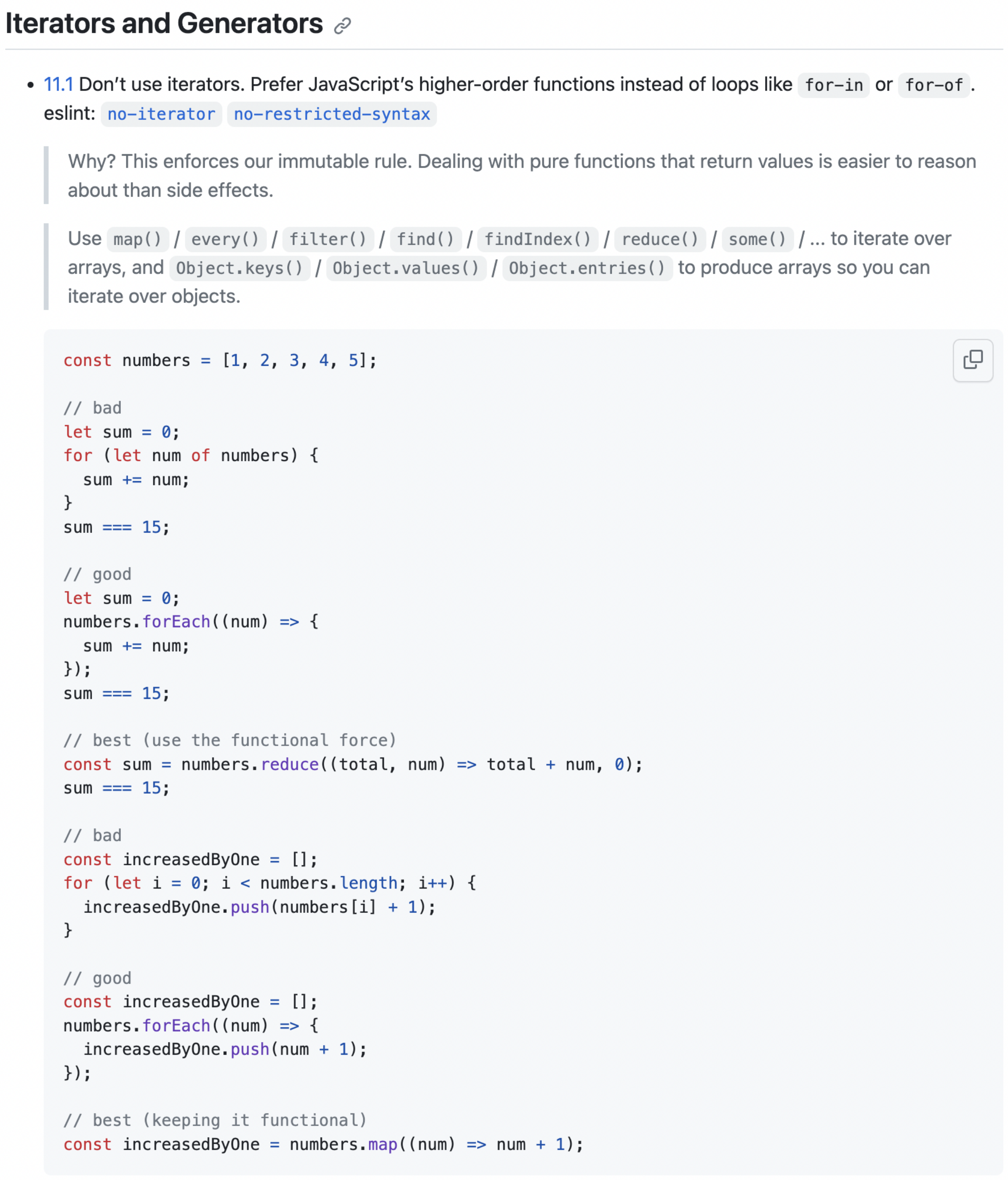
Classes & Constructors

- 9.1 Always use `class`. Avoid manipulating `prototype` directly.

  > Why? `class` syntax is more concise and easier to reason about.

```
// bad
function Queue(contents = []) {
  this.queue = [...contents];
}
Queue.prototype.pop = function () {
  const value = this.queue[0];
  this.queue.splice(0, 1);
  return value;
};

// good
class Queue {
  constructor(contents = []) {
    this.queue = [...contents];
  }
  pop() {
    const value = this.queue[0];
    this.queue.splice(0, 1);
    return value;
  }
}
```

I'm not clear on the difference between class and prototype.



Iterators and Generators

- 11.1 Don't use iterators. Prefer JavaScript's higher-order functions instead of loops like `for-in` or `for-of`. eslint: `no-iterator` `no-restricted-syntax`

  > Why? This enforces our immutable rule. Dealing with pure functions that return values is easier to reason about than side effects.

  > Use `map()` / `every()` / `filter()` / `find()` / `findIndex()` / `reduce()` / `some()` / ... to iterate over arrays, and `Object.keys()` / `Object.values()` / `Object.entries()` to produce arrays so you can iterate over objects.

```
const numbers = [1, 2, 3, 4, 5];

// bad
let sum = 0;
for (let num of numbers) {
  sum += num;
}
sum === 15;

// good
let sum = 0;
numbers.forEach((num) => {
  sum += num;
});
sum === 15;

// best (use the functional force)
const sum = numbers.reduce((total, num) => total + num, 0);
sum === 15;

// bad
const increasedByOne = [];
for (let i = 0; i < numbers.length; i++) {
  increasedByOne.push(numbers[i] + 1);
}

// good
const increasedByOne = [];
numbers.forEach((num) => {
  increasedByOne.push(num + 1);
});

// best (keeping it functional)
const increasedByOne = numbers.map((num) => num + 1);
```

My confusion here is also based on my limited understanding of the difference between loops and high order functions.