Immutable Objects

"I've always known the meaning of immutable objects, but it never quite clicked why they're so crucial in development. If you're in the same boat, then you are at right place! Discover what immutable objects are, why they're essential for robust code, and how you can start creating them today. Let's dive into the world of immutability together!

#ImmutableObjects #JavascriptDevelopment #CodePerformance

Immutable And Mutable

An immutable value is one whose content cannot be changed without creating an entirely new value. It means you can't change stuff once it's set. Imagine having a box where once you put something in, you can never swap it out. That's immutability.

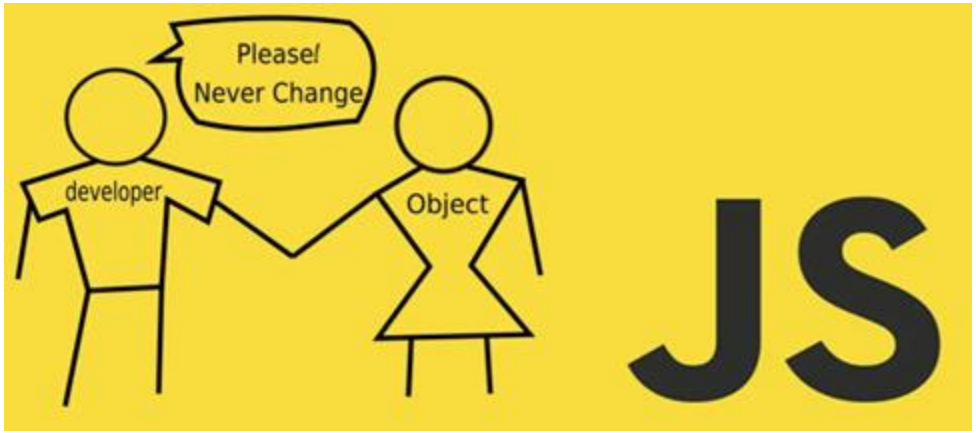
If something is "mutable," it means you can change its properties. Like, if you have a box, and you can put different things in it whenever you want, that box is mutable.

But proving something is mutable is pretty straightforward. All you need is one way to change it, and boom, it's mutable.

On the other hand, proving something is immutable is tough. You can't just rely on the programming language to guarantee it. It's more like an agreement between developers.

Immutable objects in javascript

In javascript, objects are mutable by default. It means their properties and elements can be changed without re-assigning a new value. But sometimes it is useful to use immutable objects. This is particularly useful for ensuring data integrity and preventing unintended side effects in your code.



Why we need immutable objects?...

Performance Boost: Immutable objects ensure no future changes, making programs faster by avoiding tracking potential alterations. It's like saying, "Hey, don't worry about any changes later."

Memory Saver: Rather than copying the entire object for each new version, references to it suffice, conserving memory by eliminating duplicate data storage. It's like having a blueprint and using it whenever you need, rather than building a new thing each time. This saves memory because you're not storing duplicate data.

While it might seem inefficient to create new objects every time a change is needed, JavaScript engines are optimized for this kind of operation. They use techniques like reference sharing and copy-on-write to ensure that memory usage remains efficient.

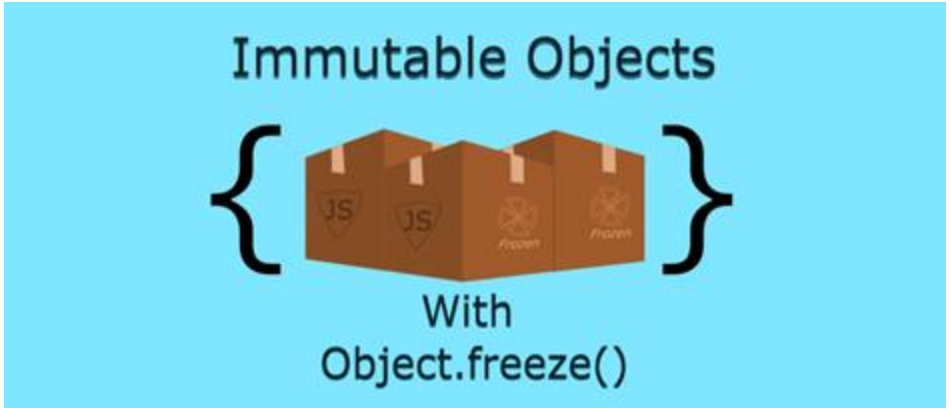
Thread-Safety: Additionally, since immutable objects cannot be changed after creation, they are inherently thread-safe, making them particularly useful in concurrent programming environments.

Developer Ease: Immutable objects offer consistent behavior, reducing developer stress by eliminating unexpected state changes, like a reliable friend who never surprises you.

How to create immutable objects in Javascript?

Mainly, there three ways as mentioned below:

1. `Object.freeze()`
2. `Spread operator`
3. `Object.assign()`



Object.freeze()

Nothing can be added to or removed from the properties set of a frozen object.

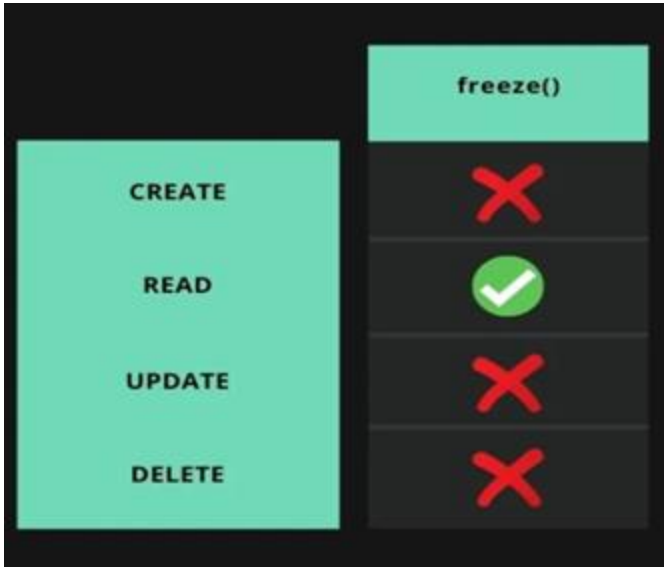
Syntax : Object.freeze(obj)

```
const obj = {
  prop: 58,
};

Object.freeze(obj);

obj.prop = 42;
// Throws an error in strict mode

console.log(obj.prop);
// Expected output: 58
```



Spread operator

The spread operator allows you to spread the elements of an iterable (such as arrays, strings, or objects), into another iterable or function call.

Syntax : { ...obj, key: 'value' }

```
const oldObj = { name: "Pranali", location: "Pune" };

//Creating new object from old one
const newObj = { ...oldObj, location: "Mumbai" };

console.log(oldObj);
//{ name: 'Pranali', location: 'Pune' }

console.log(newObj);
//{ name: 'Pranali', location: 'Mumbai' }
```

Object.assign()

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// { a: 1, b: 4, c: 5 }

console.log(source);
// { b: 4, c: 5 }

console.log(returnedTarget);
//{ a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// true
```

The Object.assign() method is used to copy the values and properties from one or more source objects to a target object.

Syntax : Object.assign(target, source1)

Properties in the target object are overwritten by properties in the sources if they have the same key.