DZone

Three Approximations You Should Never Use When Coding

by Jean-Jacques Dubray · by Marcus Feitoza · by Sladan Ristic · Jul. 20, 17 · Web Dev Zone · Analysis

The path travelled by the software industry since its inception and the accomplishments in scale and scope are simply astonishing: from creating a network the size of the planet, indexing the world's information, connecting billions of people, to driving a car or flying an aircraft, or even an application such as Excel which enables 500M people to program a computer… Turing would be amazed at what can be done with computers today and how many people use one every day, if not every minute.

In this context, innovations in software engineering have been running amok, from operating systems, runtimes, programming languages, frameworks, to methodologies, infrastructure, and operations.

However, even though "the software industry has been very successful," Ivar Jacobson and Roly Stimson report that "under the surface, everything is not as beautiful: too many failed endeavors, quality in all areas is generally too low, costs are too high, speed is too low, etc."

In particular, the evolution of programming paradigms has been slow, and we should not be afraid to say that it is currently holding back our industry. The emergence of modern languages such as Java and C#, their success, but also their unhealthy competition, have created a conceptual molasse that generations upon generations of developers cannot seem to escape. Even modern languages such as JavaScript are expected to align to their antiquated conceptual foundations from OOP to MVC.

Our goal in this article is to kick start an open discussion on the evolution of programming paradigms and offer a contribution founded on, perhaps, the most robust theory in Computer Science: TLA+.

The paper is divided into three sections:

- Computation and State Machines.
- An Advanced State Machine Model.
- A New Programming Model.

If you have not read this article which provides an introduction to TLA+ from its author, Dr. Leslie Lamport, please do. Anything we could write in comparison is just scribbles.

Computing Is About State Machines

In his book, Dr. Lamport uses the Die Hard problem to illustrate why “State Machines Provide a Framework for much of Computer Science.”

In the movie Die Hard 3, the heroes, John McClain (Bruce Willis) and Zeus (Samuel L. Jackson), must make exactly four gallons from five and three gallon jugs to prevent a bomb from exploding.

We have represented in Fig.1 the graph of possible states and actions that Dr. Lamport uses in his example. The state labels are encoded as follows: [jug_size][full,empty,partial][jug_size][full,empty,partial].

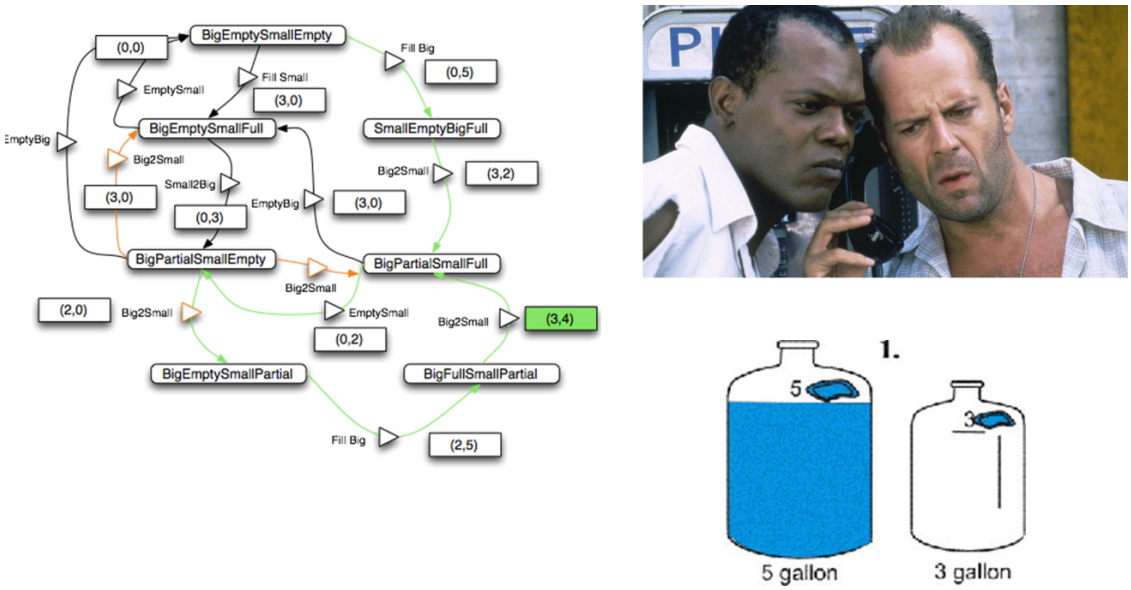


Fig. 1 – The state machine of the Die-Hard Problem and its solution (path in green)

The solution to the problem can be found by traveling a specific path.

Dr. Lamport explains that “a computation is a sequence of steps, called a behavior,” and there are three common choices for what a “step” is, leading to three different kinds of behaviors:

Action Behavior	A step is an action, an action behavior is a sequence of actions.
State Behavior	A step is a pair <Si, Si+1> of states, a state behavior is a sequence s1 → s2 → s3 → · · · of states.
State-Action Behavior	A step is a triple < Si, α, Si+1 > where S are states and α an action

From the state machine from Fig. 1, we can define an “action behavior” as a series of actions:

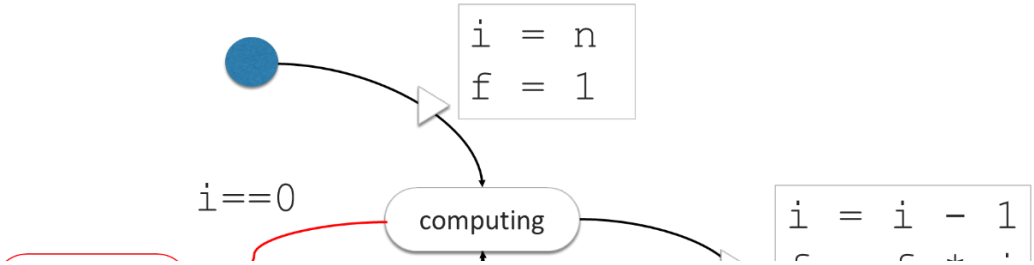
1. Fill Big.
2. Fill Small from Big.
3. Empty Small.
4. Fill small from Big.
5. Fill Big.
6. Fill Small from Big.

Incidentally, we just created a new “programming language” based on the “Die-Hard” state machine. We don’t expect it will become popular anytime soon, but in essence, that is how action-based programming languages are created: there is an underlying state machine from which actions are derived, which then can be composed any way we wish without paying much attention to the underlying state machine. Programming languages are “action based” since the vast majority of “states” in our programs are irrelevant.

Let’s go the other way and derive the underlying state machine from a code snippet, for instance, the factorial function (From Dr. Lamport’s article):

```
var fact = function(n) {

    return (n == 1) ? 1 : n * fact(n-1);

}


var fact = function(n)  {

    var f = 1, i =1 ;

    for (i = 1; i <= n; ++i) { f = i * f; }

    return f ;

}
```

We can infer a representation of the corresponding (classical) state machine for each function:



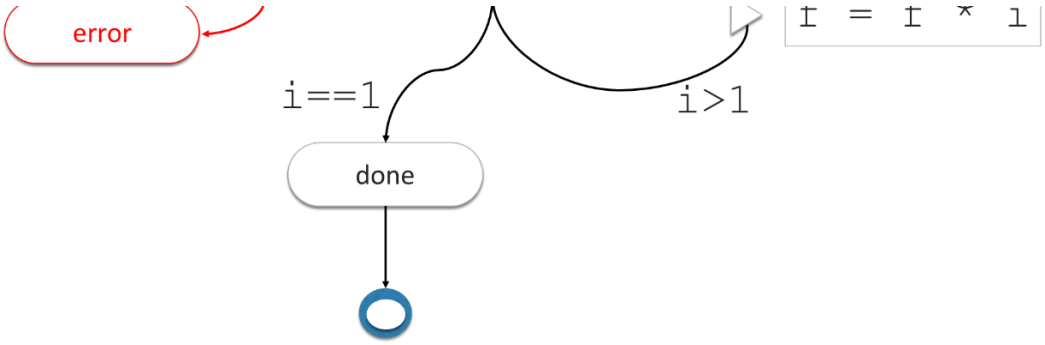


Fig. 2a – The State Machine of the Factorial Function

The behavior of the State Machine is straightforward, we initialize it with the values of i (input) and f (output) and the state machine will transition to the computing "state" until i is equal to 1, at that point f contains the expected result.

The same could be done for the recursive version. You will notice that the memory requirements are different as the state machine needs to be designed to keep at least one intermediate value (fi-1). You may also decide to keep all intermediate values to speed up future computation.

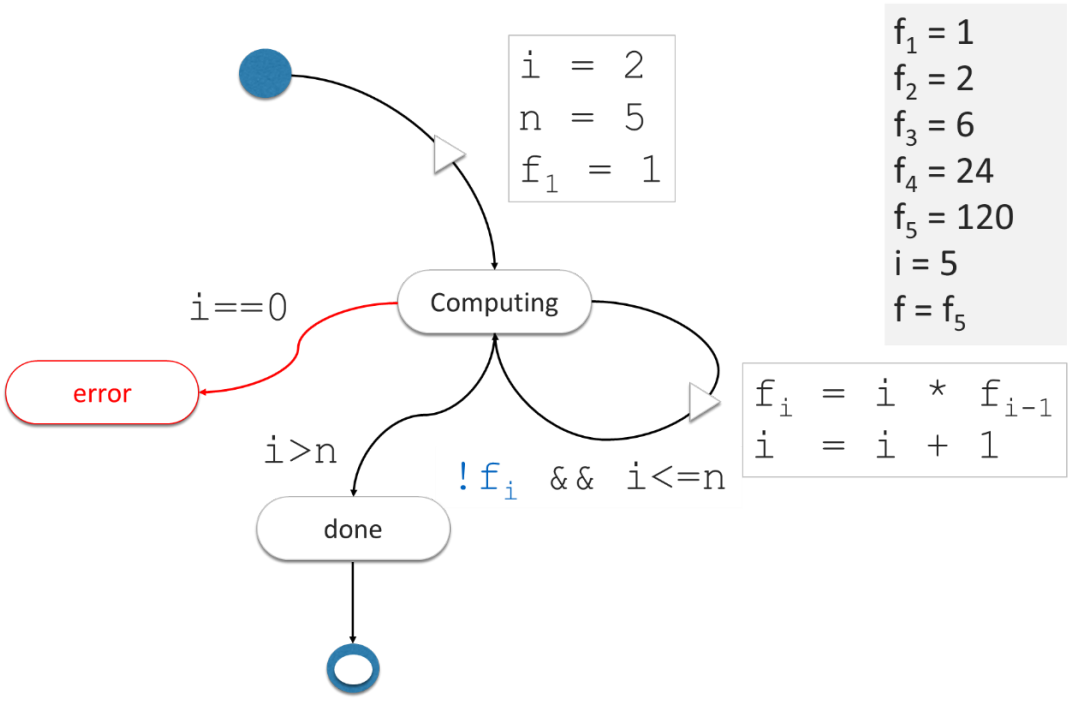


Fig. 2b – The State Machine of the Factorial Function (Recursive)

In this first section, we have demonstrated that state machines can compute and conversely (action-based) computations can be represented as state machines, but for the most part, states of the state machine are redundant, merely the shadow of actions. In the factorial functions, the (control) states (computing, done, error) would not add any clarity to the implementation. That is why for the vast majority of programs we never bother making the states explicit, and there is no argument there.

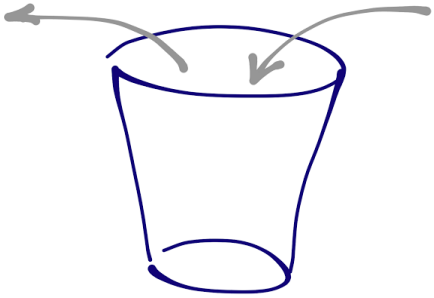
A New Kind of State Machines Based on TLA+ Semantics

There is, however, a problem with "action-Based" formalisms and programming languages. It can be illustrated with a simple observation using a glass of water.

In an action-based behavior, this system would be described by a state variable v (volume of water in the glass) and two actions: add water, drink.

On the other hand, the state-action behavior of the glass relies on three control states, and the corresponding next-actions:

- full: drink.
- in-between: drink, add water.
- empty: add water.



We apologize for the triviality of this example, but we must come to the realization that in an action-based programming model the concept of “control state” will have to be emulated in some ways, or the system will not be able to respond to actions correctly (fill the glass infinitely, drink from a negative value).

In structured programming, correctness is achieved with the use of conditional expressions: “if-then-else” or “switch-case,” which tend to focus the attention of the developer on what happens next, i.e. the sequence of actions, without requiring the precise identification of (control) states, i.e. where the system is currently at, and consequently without explicitly specifying the actions that are enabled in each (control) state.

To some extent, even state-action behaviors are challenged to provide a formalism where “control states” are modeled properly because the classical State Machine metamodel connects the action to the end state, as if, the action of adding water to the glass would be able to know when the glass is full.

When you think of it, it must be the target of the action which should decide on the resulting (control) state when applying an action. Actions should have no knowledge whatsoever of the “current state” (let alone current control state) of its target. An action should only present a series of variable assignments to its target which may or may not accept them, and, as a result, reach a new control state (or not). This seemingly mundane argument is perhaps why writing code is so error prone, because we use two approximations: either we ignore the (control) states and let actions assign variables directly, or, when (control) states are explicit, the classical state machine semantics enables the actions to decide which control state should be reached. It is possible to write code using these two approximations because in some cases, this works, but they are nevertheless approximations and cannot be generalized to every use case.

TLA+ introduces new semantics that defines a new structure for state machines that we feel can be used to write code without relying on these approximations. TLA stands for the Temporal Logic of Action and

to write code without relying on these approximations. TLA stands for the Temporal Logic of Action and TLA+ uses simple mathematics to specify computations. TLA+ is a formal specification language which is often used to test the correctness of the most challenging algorithms.

The TLA+ specification of the factorial algorithm is as follows:

$$
\begin{aligned}
Init &\triangleq (f = 1) \land (pc = \text{“test”}) \\
Next &\triangleq \quad (\quad (pc = \text{“test”}) \\
&\qquad \land (pc' = \text{IF } i > 1 \text{ THEN “mult” ELSE “done”}) \\
&\qquad \land (f' = f) \land (i' = i)) \\
&\quad \lor (\quad (pc = \text{“mult”}) \\
&\qquad \land (pc' = \text{“test”}) \\
&\qquad \land (f' = i * f) \\
&\qquad \land (i' = i - 1))
\end{aligned}
$$

Fig. 4 – The TLA+ Specification of the Factorial Algorithm

Init is a state predicate which initializes the state machine and next is a transition predicate which specifies the behavior of the state machine. The transition predicate is composed of disjunct formulas, each describing a piece of code that generates a step.

You will note that:

- The (application) “state” is a series of property values (not to be confused with control state).
- In reference to Von Neumann, Dr. Lamport often uses the “pc” variable (program counter) to store the control state of the state machine.
- Variables are primed or unprimed. The “prime” notation indicates the property values of the (application) state after the current step completes.
- An action is an ordinary mathematical formula that contains primed as well as unprimed variables (in this case, the transition predicate is composed of two Actions).

We are now in the position of redefining a new set of semantics for state machines based on TLA+.

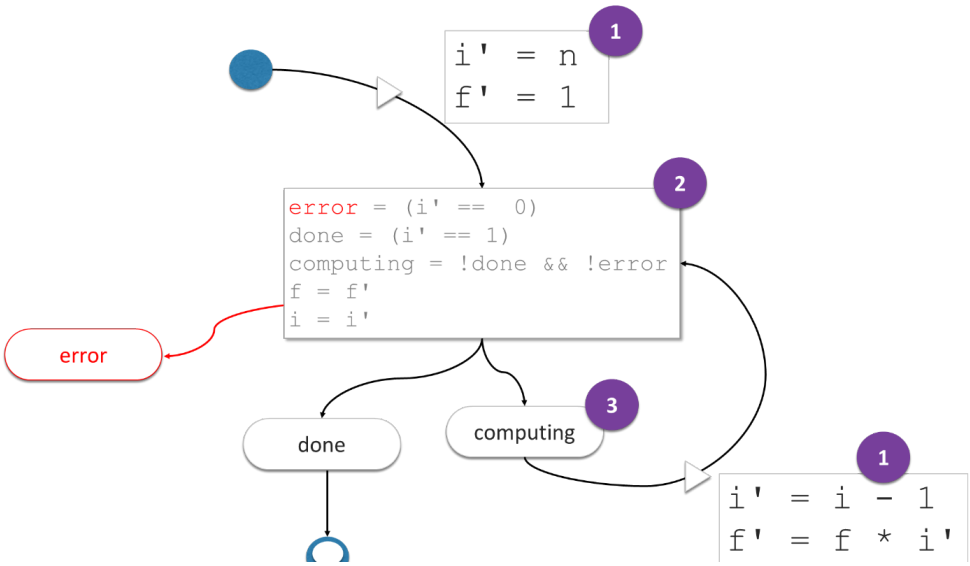


Fig. 5 – A new kind of State Machine based on TLA+ Semantics

In this state machine, the actions compute the next state property values (1), then the (application) state is updated (2), including the control state (as property values) and the transition predicate decides (3) if another step is needed (based on the control state of the state machine).

This new State Machine structure allows us to write code without the two approximations that we have identified earlier: the actions are never in the position to choose which resulting (control) state will occur from their effect. Hence, the relationship between actions and (control) states is simply observed. This approach also corrects the second approximation since actions do not mutate the property values of the state, it merely proposes new property values which will be used to compute the final state of the step.

Actually, this new structure corrects the third approximation that Dr. Lamport identified when asking the question: what is a programming step?

How many steps are performed in executing the statement f = i ∗ f ?

- Does a single step change the value of f to the product of the current values of i and f?
- Is the evaluation of i ∗ f and the assignment to f two steps?
- Are there three steps—the reading of i, the reading of f, and the assignment to f?

When we write code, we generally think of a line of code as being a programming step, but this could not be further from the truth. A step here is an action, a mutation, and a resulting (control) state.

The SAM Pattern

The State-Action-Model Pattern (SAM) was introduced by Jean-Jacques Dubray in June 2015 and became popular throughout 2016. As a preamble to this section, we would like to change the meaning of the word "State." Even though these semantics are well defined and widely accepted in the software industry, they encourage the confusion of ignoring the (control) states of State Machines, which are always reified as properties of the so called (application) "State" (including in TLA+). We would like to suggest instead that we call the set of property values (unprimed variables) of a system, the "Model" and reserve the word "State" for control States, as it is used in classical State Machines.

We can then map the SAM Pattern to TLA+ as shown in Fig. 6.

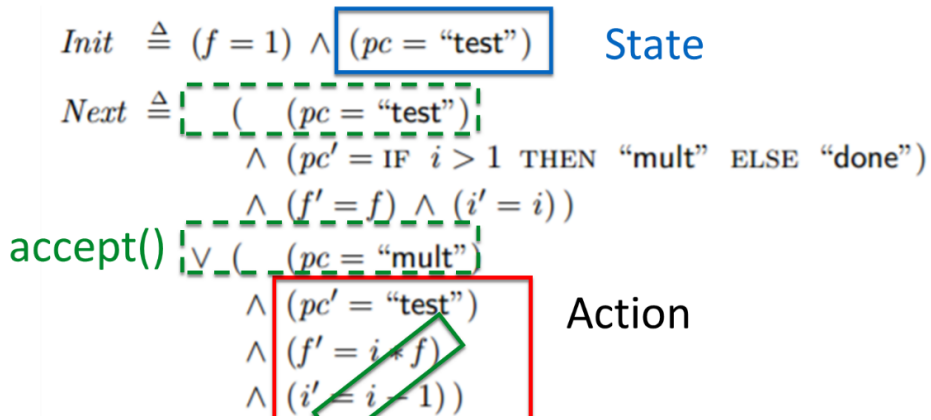


Fig. 6 – State, Action, Model mapping to TLA+

State	Action	Model
pc	f' = i * f i' = i - 1	f,i In TLA+ assignments, primed variables to non-primed variables are implicit but controlled via logic predicates.

The SAM Pattern corrects the three approximations that we have identified in the previous section, with SAM:

- Actions cannot mutate the properties of the model.
- Assignments are not mutations, mutations must be explicit.
- Action→Model→State specify the "programming step," a line of code is not a step.

In SAM, Actions are functions which input is an event and output is a proposal (primed variables) to mutate the property values of the Model (unprimed variable). The Model is an instance of a class (or a closure) generally with a single method which is used to receive proposals to mutate the property values. When the mutation is complete, the model invokes the "State" function whose role is to compute the (control) state, which can also be viewed as the state representation of the model. Last but not least, the State function invokes the next-action predicate which computes whether an action needs to be triggered in the current (new) occurrence of a control state. The next-action predicate will return true if one action was triggered and false otherwise.

There are multiple ways to implement the SAM pattern. A "reactive loop" is generally preferred. In its simplest form, in JavaScript, for instance, the pattern would look like this code snippet:

```
let actions = {

    countBy(step) {

        step = step || 1
            model.accept({incrementBy:step})

    }

}

```

```
let model = {

     counter: 0,

     accept(proposal) {

         this.counter += (proposal.incrementBy>0) ? proposal.incrementBy : 0

         state.render(this)

     }

}

let state = {

   render(model) {

       // render state representation
       if (!this.nextAction(model)) { console.log(model.counter) }

   },


   nextAction(model) {

     if (model.counter %2 === 0) {
        // let's pretend there is a next action
        actions.countBy(1)
        // do not render state
        return true
     } else {
        // no next action, render state
        return false
     }
  }
}


// invoke actions to start the reactive loop
actions.countBy(2)
actions.countBy(5)
```

Fig. 7 – The SAM Implementation of a Counter in JavaScript

In this example, the Reactive Loop is enacted by an event triggering the countByOne() action, continues with the accept method of the model, and then the State render function. At a high level, you can start

with the accept method of the model, and then the State render function. At a high level, you can start using SAM today if you follow this simple recipe: each time you need to write an event handler, you split your code into three buckets: action/model/state.

- Actions are responsible for validating, transforming, and enriching events.
- The Model is responsible for mutating the property values (aka the application state).
- The State responsible for computing the State Representation (aka the control state) that other parts of your application will want to consume.

Actions can be grouped into two different categories: idempotent and non-idempotent. Idempotent actions result in a constant state representation, no matter when or how many times they are applied. In REST terms, a path to a resource should rather be viewed as an idempotent action which, when triggered, moves the system to a constant state.

One of the key advantages of SAM is that you no longer need ancillary state machines to manage side effects. Actions can call APIs and propose the results to the model, there is no particular reason to enter the model and mutate a property such as “fetching,” just to keep track of the API call lifecycle.

It’s also easy for each state representation, to keep track of “allowed actions,” such that when an action is triggered it should result in an error condition or if it should proceed to create a proposal. The SAM-SAFE library was designed to illustrate that aspect.

The semantics of the SAM pattern are simple, but not simplistic. They are derived from TLA+, perhaps the most robust theory in Computer Science today. Even though there might be some specific optimizations in a given context, you should proceed with great care when you feel some semantics would fit better. You should contrast this approach with the history of programming languages and how their semantics came about from Grace Hopper to Elm. Very few, if any, of the programming languages, were founded on Computer Science itself. For instance, “Object Oriented Programming grew out of Doug Englebart's vision of the computer as an extension of the human mind and the goal of object-oriented programming pioneers [such as Alan Kay] was to capture end user mental models in the code.” Mark Allen reports that the foundation of Cobol was designed by a committee, in six months, with the goal of creating a portable programming language (across proprietary platforms at the time). As a matter of fact, programming languages are either built on a “functional foundation” (from Fortran and on) or a “Data Type” foundation (starting with Lisp). Let’s assume you knew nothing about programming, and someone would explain to you what programming is, and continue by saying the fundamental building block of programming is: a) a pure function or b) a class, wouldn’t you ask the question why? Have you ever asked that question? Is there a chance these statements could be fallacies?

Conclusion

Since the first structured programming language and its compiler were created, software engineering has been using the same three approximations. These are not only unnecessary but possibly make it harder to write correct software. The goal of this paper was to identify these approximations, offer a new path to software engineering based on TLA+ semantics, and illustrate how simple it is to write code without using these approximations.

Since the development of the SAM pattern in the spring of 2015, several big projects have been developed on that foundation demonstrating its benefits and robustness.

Many thanks to Mark Allen and Jouko Salonen for reviewing this article.

Additional references:

- SAM Pattern: http://sam.js.org
- SAM Repository: https://github.com/jdubray/sam-samples
- SAM SAFE Library: https://www.npmjs.com/package/sam-safe
- API Orchestration with SAM: https://bitbucket.org/jdubray/star-javascript, https://bitbucket.org/jdubray/star-java, https://github.com/robsiera/StarCsharp

Some of the content in this article was given during a Lecture at Mae Fah Luang University, Computer Science Department, by Jean-Jacques Dubray

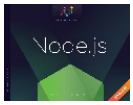
---

Like This Article? Read More From DZone

Difference Between var, let, and const Keywords in JavaScript

List of Dos and Don'ts in JavaScript Development

How JavaScript Actually Works: Part 1



Free DZone Refcard
Node.js

Topics: JAVASCRIPT BEST PRACTICE , SOFTWARE ENGINEERING , PATTERNS , WEB DEV

st in Production (the Netflix Way)
unchDarkly