# What is Scope

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values...

In fact, the ability to store values and pull values out of variables is what gives a program `state`

Without such a concept, a program could perform some tasks, but they would be extremely limited and not terribly interesting **But** where do these variables live?

These questions speak to the need for a well-defined set of rules for storing variables in some location, and for finding those variables at a later time...

**We'll call that set of rules:**`scope`

But, where and how do these *scope* rules get set?

## Understanding Scope

The way we will approach learning about scope is to think of the process in terms of a conversation **BUT WHO,** is having this conversation?

Let's meet the cast of characters that interact to process the program: `let a = 2`

1. *Engine*
    - Responsible for start-to-finish compilation and execution of our JavaScript program
2. *Compiler*
    - One of Engine's friends; handles all the dirty work of parsing and code-generation
3. *Scope*
    - Another friend of Engine; collects and maintains a look-up list of all the declared identifiers (variables)
    - and enforces a strict set of rules asw to how these are accessible to currently executing code

We need to **fully understand** how JavaScript works... you need to begin to *think* like Engine (and friends) think, ask the questions they ask, and answer those questions the same.

When you see the program `let a = 2;` you most likely think of that as one statement But that's not how Engine sees it... In fact, Engine sees two distinct statements

1. one that Compiler will handle during compilation
2. one that Engine will handle during execution

So, let's break down how Engine and friends will approach the program: `let a = 2;`

The first thing Compiler will do with this program is perform **lexing** to break it down into tokens, which it will then prase into a tree

But when Compiler gets to code generation, it will treat this program somewhat differently than perhaps assumed

A reasonable assumption would be that Compiler will produce code that could be summed up by this pseudocode: "Allocate memory for a variable, label it `a`, then stick the value `2` into that variable" Unfortunately, that's not quite accurate...

Compiler will instead proceed as:

1. Encountering `var a`, Compiler asks Scope to see if a variable `a` already exists for that particular scope collection. If so, Compiler ignores this declaration and moves on. Otherwise, Compiler asks Scope to declare a new variable called a for that scope collection
2. Compiler then produces code for Engine to later execute, to handle the `a = 2` assignment. The code Engine runs will first ask Scope if there is a variable called `a` accessible in the current scope collection. If so, Engine uses that variable. If not, Engine looks `else-where`

If Engine eventually finds a variable, it assigns the value `2` to it. If not, Engine will raise its hand and yell out an error!

To summarize: two distinct actions are taken for a variable assignment: First, Compiler declares a variable (if not previously declared) in the current scope, Second, when executing, Engine looks up the variable in Scope and assigns to it, if found.

## Engine/Scope Conversation

```
function foo(a) {
  console.log(a); // 2
}

foo(2);
```

Let's imagine the above exchange (which processes this code snippet) **as a** conversation...

Engine: Hey Scope, I have an RHS reference for `foo`. Ever heard of it? Scope: Why yes, I have. Compiler declared it just a second ago. It's a function Engine: Great, thanks! Ok, I'm executing `foo` Engine: Hey Scope, I've got a LHS reference for `a`, ever heard of it? Scope: Yup, Compiler declared it as a formal parameter to `foo` Engine: Great, now time to assign `2` to `a` Engine: Hey Scope, I need a RHS look-up for `console` Scope: No problem, I've got `console` it's built in Engine: Perfect, Looking up `log(..)`... Great it's a function Engine: Can you help me out with a RHS reference to `a`, I think I remember it but not sure... Scope: You're right, Engine. Same variable, hasn't changed Engine: Cool, passing the value of `a`, which is `2`, into `log(..)`

### Quiz

```
function foo(a) {
  let b = a;
  return a + b;
}
```

```
let c = foo(2);
```

1. Identify all the LHS lookups (there are 3)
2. Identify all the RHS lookups (there are 4)

Answers:

All LHS lookups: `c=..`, `a = 2 (implicit param assignment)`, and `b=...` All RHS lookips: `foo(2..)`, `= a`, `a..` and `..b` (return statement)

## Nested Scope

Scope is a set of rules for looking up variables by their identifier name There's usually more than one scope to consider...

Just as a block or function is nested inside another block or function, scopres are nested inside other scopes So, **if a variable cannot be fonud in the immediate scope**... Engine consults the **next** outercontaining scope, **continuing** until it is found or until the **outermost (aka global) scope has been reached**

consider the following:

```
function foo(a) {
  console.log(a+b);
}

let b = 2;
foo(2); // 4
```

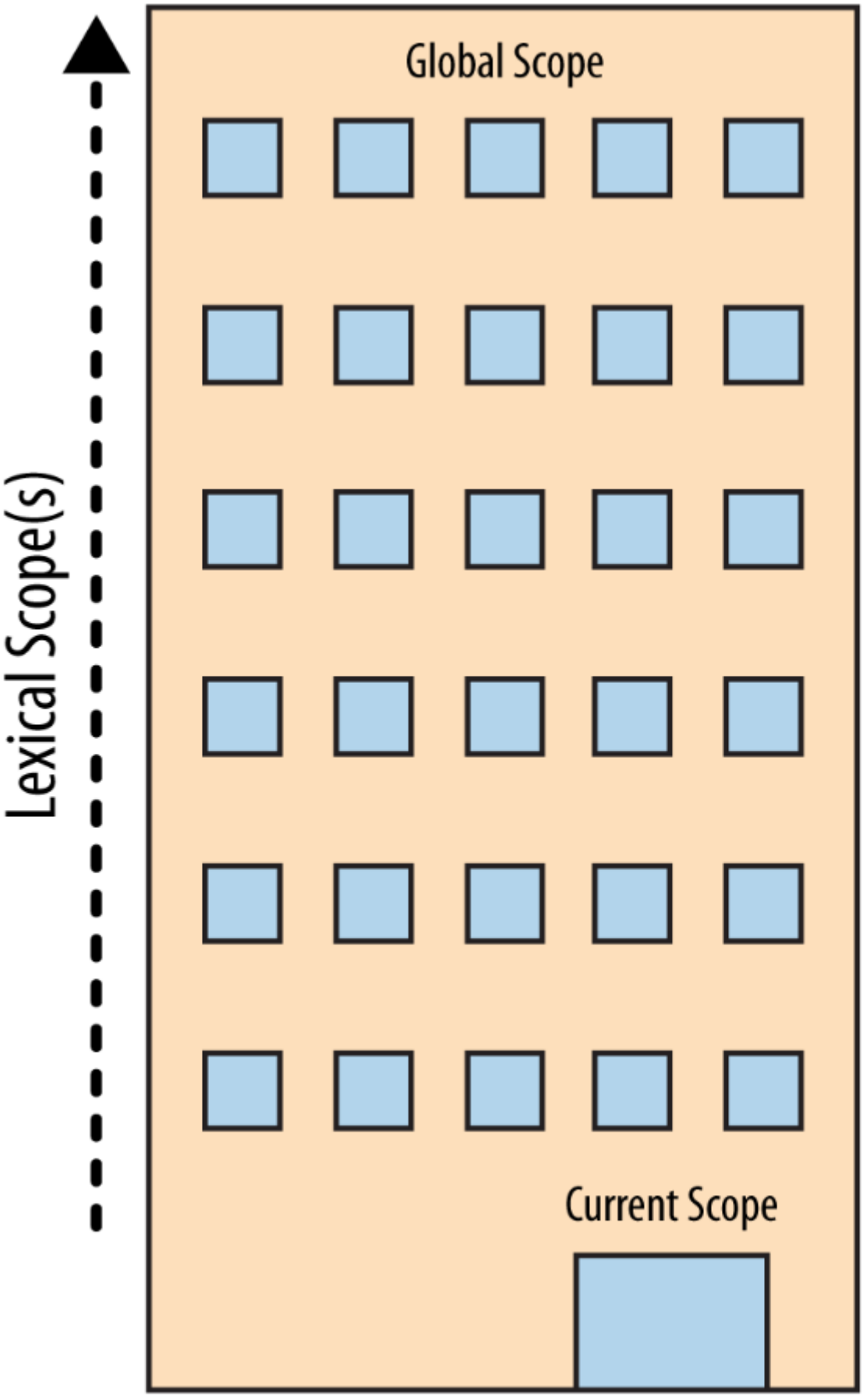
The RHS reference for `b` cannot be resolved inside the function `foo`, but it can be resolved in the scope surrounding it **In this case it was the `global` scope**

So, revisiting the conversation between Engine and Scope, we'd overhear

> Engine: “Hey, Scope of foo , ever heard of b ? Got an RHS reference for it.” Scope: “Nope, never heard of it. Go fish.” Engine: “Hey, Scope outside of foo , oh you’re the global scope, OK cool. Ever heard of b ? Got an RHS reference for it.” Scope: “Yep, sure have. Here ya go.”

The simple rules for traversing nested scope: Engine starts at the currently executing scope, looks for the variable there, then if not found... **keeps going up one level**, and so on If the outermost global scope is reached, the search stops, **whether it finds the variable or not**

To visualize the process of nested scope resolution we have this:




The building represents our program's nested scope ruleset. The **first floor** of the building represents your currently executing scope, wherever you are The **top level** of the building is the global scope

You resolve LHS and RHS references by looking on your current floor, and **if you don't find it**, take the elevator to the next floor... looking there, then the next, and so on... Once you get to the top floor(**the global scope**), you either find what you're looking for **or** you don't But you have to stop regardless

## Review

Scope is the set of rules that determines where and how a variable can be looked up

This look-up may be for the purposes of assigning to the variable, which is an LHS (lefthand-side) reference **or** it may be for the purposes of retrieving its value, which an RHS reference

**LHS** references result from assignment operations Scope-related assignmnets can occur either with the `=` operator **or** by passing arguments to (assign to) function paratmeters

The JavaScript engine first compiles code before it executes, and in so doing... it splits up statements like `var a = 2`

1. First, `var a` to delcare it in that scope. This is performed at the beginning, before code execution
2. Later, `a = 2` to look up the variable (LHS reference) and assign to it if found

Both LHS and RHS reference look-ups start at the currently executing scope, **and if need be** (that is, they don't find what they're looking for there), they work their way up the nested scope, one scope(floor) at a time, looking for the identifier, until they get to the global(top floor) and stop, and either find it, or don't

Unfulfilled RHS references result in `ReferenceErrors` being thrown Unfulfilled LHS references result in an automatic, implicitly created global of that name (if not in Strict mode), or a `ReferenceError` (if in Strict Mode)

## Lexical Scope

There are two predominant models for how scope works The first first of these is by far the most common: *Lexical Scope* The other model, is only used by **some** programming languages(such as Bash scripting) is called *dynamic scope*

If you recall, the **lexing process** examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing

It is this concept that provides the foundation to understand what lexical scope is and where the name comes from

To define it somewhat circularly, **lexical scope is** is scope that is defined at lexing time

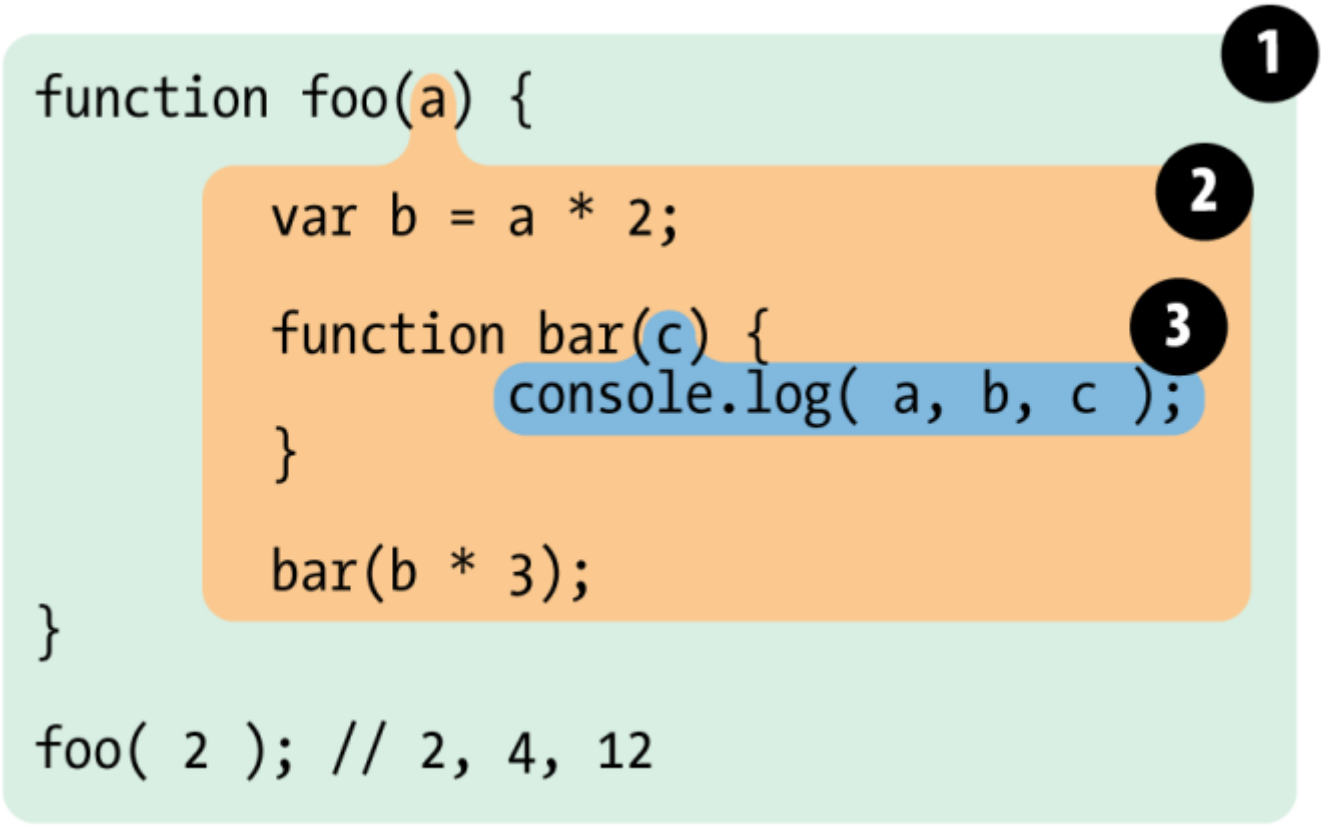
In other words, lexical scope is based on **where the variables and blocks of scope are authored**, by you, at write time, and this is set in stone by the time the lexer processes your code

Let's consider this block of code:

```
function foo(a) {
  let b = a * 2;

  let bar(c) {
```

```
    console.log(a,b,c);
  }
  bar(b*3);
}

foo(2); // 2, 4, 12
```

There are three nested scopes inherent in this code example.




Bubble 1 encompasses the global scope and has just 1 identifier in it: `foo`

Bubble 2 encompasses the scope of `foo`, which inclides the three identifiers: `a`, `bar`, and `b`

Bubble 3 encompasses the scope of `bar`, and it includes just one identifier: `c`

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other

For now, let's just assume that each function creates a new bubble of scope

The bubble for bar is entirely contained within the bubble for `foo`, because that's where we chose to define the fnuction `bar`

Notice that these nested bubbles are strictly nested We're not talking abot Venn Diagrams where the bubbles can across boundaries In other words, no bubble for some function can simultaneously exist inside two other outer scope bubbles, just as no function can partially be inside of each two parent functions

## Lookups

**No matter** *where* a function invoked from, or even *how* it is invoked, its lexical scope is *only* defined by where the function was declared

The lexical scope look-up process *only* applies to the first-class identifiers, such as the `a`,`b`,`c` If you had a reference to `foo.bar.baz` in a piece of code, the lexical scope look-up would apply to finding the `foo` indentifier, but once it lovates that variable, object property-access rules take over to resolve the `bar` and `baz` properties, respectively

## Reviewing Lexical Scope

Lexical Scope means that scope is defined by author-time deicisions of where functions are declared

The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked up during execution

Two mechanisms is JavaScript can "cheat" lexical scope: `eval(..)` and `with` The former can modify existing lexical scope (at runtime) by evaluating a string of "code" that has one or more declarations in it

The latter essentially creates a whole new lexical scope by treating an object reference *as* a scope and that object's properties as scoped identifiers

The downside to these mechanisms is that it defeats the engine's ability to perform compile-time optimizations regarding scope look-up, because the engine has to assume pessimitically that such optimizations will be invalid Code *will* run slower as a result of using either feature. **Don't use them**.

---

Scopes from the videos

# Deeper Understanding of Scopes in JavaScript

If we have colored buckets and colored marbles, naturally we want to match each with its corresponding color

But in the our scenario, what are the buckets? and what are the marbles? The buckets are our units of scope and the marbles are our identifiers

Functions and blocks are our units of scopes...

```
let teacher = "Kyle";

function otherClass() {
  let teacher = "Suzy";
  console.log("Welcome");
}

function ask() {
  let question = "why";
  console.log(question);
}
```

```
otherClass(); // Welcome
ask(); // Why?
```

We have a formal declaration `teacher = "Kyle"` and our global scope recognizes and stores it (red marble - teacher) and (red bucket - global scope)

Then we have `otherClass()`, we formally declare this function and scope recognizes it and stores it (red marble as well, bc its an indentifier in our global scope). So 2 red marbles in the red bucket

**But** the compiler realizes that its a special kind of marble, because it creates a scope So now we have a new bucket inside our red bucket... our new bucket is blue

Inside our function `otherClass()` we have another formal declaration `let teacher = "Suzy` So our new scope manager for blue bucket has never heard of this marble `teacher` so we put a new marble in the blue bucket No more formal declarations in `otherClass()` so we step outside of the blue bucket and are back in the red bucket

**remember** red bucket is global scope, and blue bucket is only the scope for our function `otherClass()` Once we step outside of the bluc bucket we're back in the red bucket

As we continue parsing/compiling our code, we notice another formal identifier the function `ask()` Since we encountered this in the global scope, this identifier is a red marble that belongs in the red bucket **but again**, this is a special marble bc it creates its own scope and therefore a new bucket, lets say our new bucket is color green

We step inside our green bucket, and we look for formal declarations. And we find it... `let question = "Why"`, has the scope encountered this identifier before? No, so we add a new green marble to our green bucket After searching for more declarations, we don't find any and step outside of the scope **and back to** the global scope

Now we are done, because there are no more formal declarations, just two function invocations **BUT**, these invocations have nothing to do with the created scopes

**SO**, one key takeaway is taht in a lexically scoped language which JavaScript is... All of the scopes that we're dealing with, all of the lexical scopes and identifiers, that's all **determined at compile time** It is **not** determined at run time, It is **used** at run time, **but** it is determined at compile time

And why that matters is, that allows the engine to more effectively optimize, because everything is known and it's fixed Nothing during the run time can determine that this marble is no longer red, now it's blue Once we've processed through we know what color marbles we're using, and the JavaScript engine can be more efficient

## Executing code

`let teacher = "Kyle";` this declaration is actually two separate things

1. The `var teacher` part, that's what the compiler handles
2. The `teacher = "Kyle"` part, that's what the execution engine is going to handle

`ask()` this is a source reference for a marble called `ask` Hey red scope, I have a marble called ask, and scope hands it to JS engine And this marble introduces us to the green bucket We has the scope in the green bucket what is in this scope and we execute the code inside

## Lexical Scope Review

We recall that JavaScript is **not** an interpreted language and that it goes step by step one line at a time

But rather we should think about JavaScript as a two pass processing

First pass, we could call it compilation or pasing, but we're going to go through the entire code And there's lots of things that happen during the parsing and compilation, but the main thing that happens is... all of the scopes, all of those colored marble buckets, the plan for those all get created, we figure out where all the scope boundaries are And indeed, all of the identifier references, those are color coded as marbles So we'll have a red marble or a blue marble or a green marble

And we'll use that information about the color of the marble and what bucket it comes from, that information is critical and useful in the second pass when we execute code

In the compilation phase, remember it's the compiler and the scope manager or the parser, whatever you want to think of **and** the scope manager They're going to be talking to each other

```
let teacher = "Kyle";

function otherClass() {
  let teacher = "Suzy";
  console.log("Welcome");
}

function ask() {
  let question = "why";
  console.log(question);
}

otherClass(); // Welcome
ask(); // Why?
```

So the compiler enters this code and asks the scope manager (we're talking to the global scope now) I have a formal declaration for a variable called `teacher`, and have you ever heard of it before (remember red bucket == global scope) The scope manager says no, and proceeds to create a red marble `teacher` and places it in the red bucket

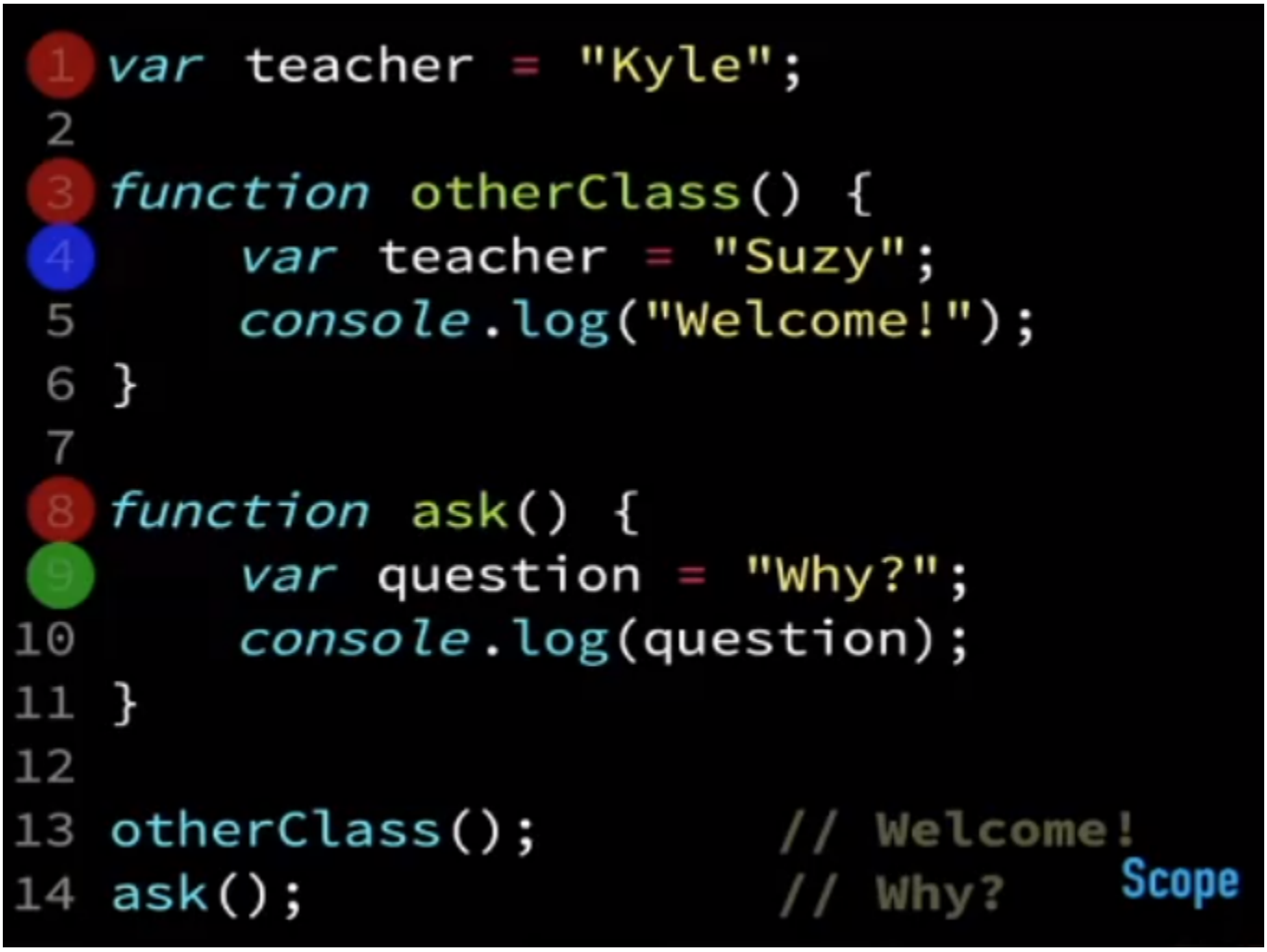
Then we find another formal declaration `otherClass()`, the compiler asks the global scope if it has ever heard of it.. to which the scope manager says no and proceeds to create a red marble called `otherClass` and places it in the red bucket

However, this red marble is special because it points to a function so we need a new bucket (scope)... so the global scope creates a new blue bucket(scope) inside of our red bucket, and the compiler enters this scope and finds a new formal declaration The compiler then asks the **blue scope manager** if it's heard of the

identifier called `teacher` to which the scope manager says no and creates a blue marble called teacher (the blue scope manager hasn't heard of `teacher` because we're in a new scope/bucket) After finding no more identifiers in the scope, the compiler returns to the global scope

Then the compiler finds another declaration, and asks the scope manager (of the global scope) if it's heard of `ask`, to which the scope manager says no and proceeds to create another red marble called `ask()` This marble points to a function and therefore the scope manager creates another bucket(color green) The compiler enters this scope and looks for more declarations to which it fins a formal declaration called `question` Compiler asks the **green scope manager** if its heard of `question` to which the scope manager says no and proceeds to create a green marble Then places the green marble in the green bucket and exits this scope after finding no more declarations

Then the compiler is back in the global scope and finds no more formal declarations




So now we're going to say we are now the execution engine, the VM We are going to execute this code and we are going to say, hey scope manager (hey global scope) I have a **target reference** for teacher, ever heard of it? The scope manager says yes, because there is one in the scope, so here's your red marble I go to the right hand side, I find the value, I assign it in the target, and we're done with the first line

Now **execution** is going to move to line 13, because remember, we're only reading executable code Hey global scope I have a **source reference**, because it's getting values, but also because its simply not a target reference If it is not receiving a target, it must be a source, because those are the only two options

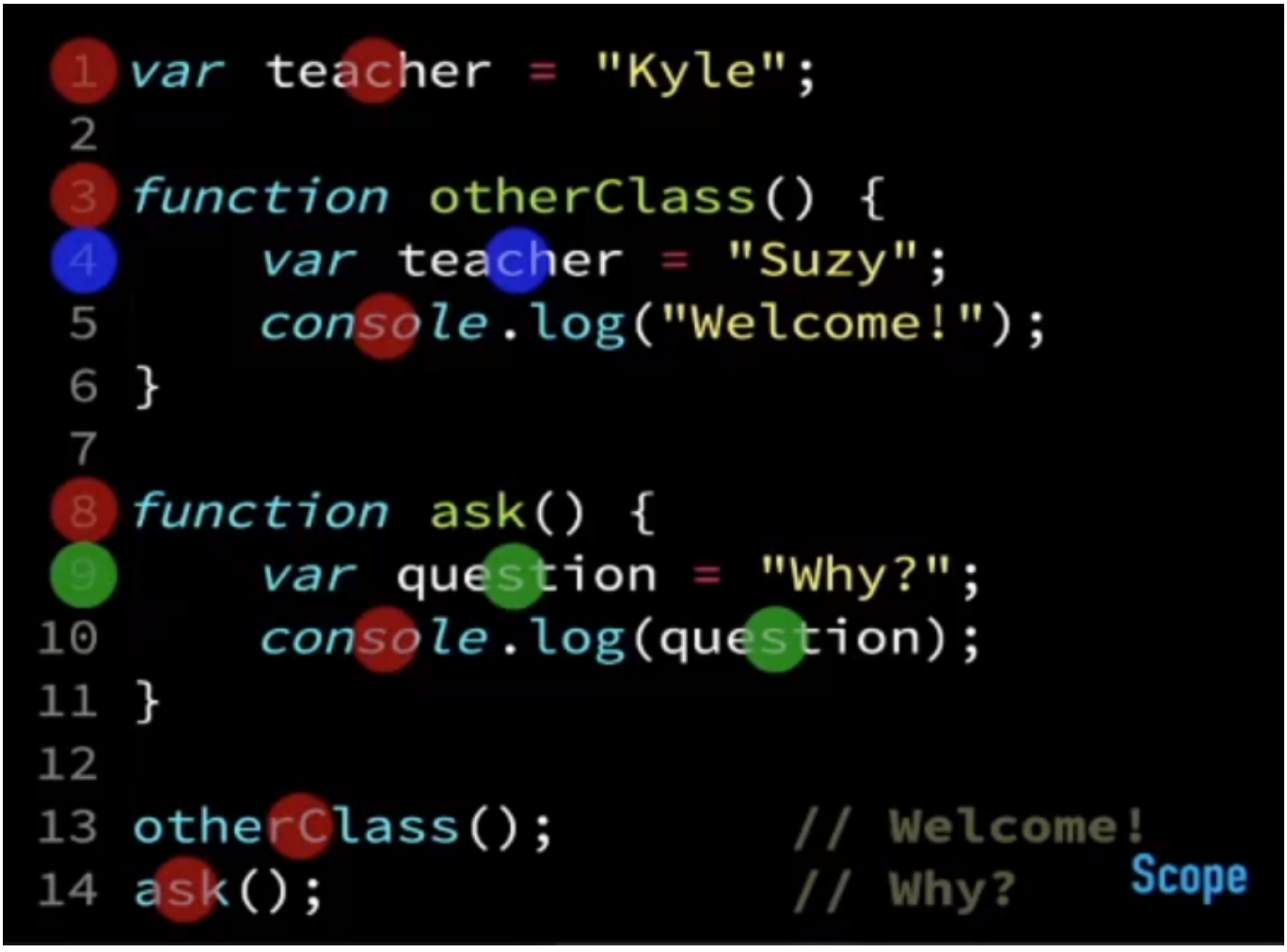
So hey global scope i have source reference for another class, have you heard of it global scope says yes and hands compiler the red marble we go and look what's in the red marble, conceptually, what's in that

area in memory, what's it holding? a reference to the "blue bucket"

So now we're going to execute that, hey blue scope I have a target reference for teacher, and the blue scope manager hands us the blue marble for teacher we go to the right hand side and assign it and we're done with that line Then we have a source reference for console, hey scope of blue bucket have you ever heard of console? We go up 1 level and ask global scope if they have heard of it, to which they have, and they hand us the marble that allows us to access the console function.

Then we continue, the engine has a source reference for `ask()`. Scope manager hands us the marble for `ask` and we look what's in the red marble Again **this source reference** holds a reference to the green bucket, the scope of the function `ask()`

So now we're going to execute that, hey green scope, I have a target reference for `question`, the scope manager gives us that marble We go to the right hand side grab the value and assign it to the target. Then we ask the scope manager for the console marble, but green scope manager doesnt have it, but the global scope does have that marble Once receiving it we can excute the console.log function




**IF** you're using transpiled code (like `babel`), `strict mode` is definitely on

It's important to remember that **function declarations make their identifier in their enclosing scope**

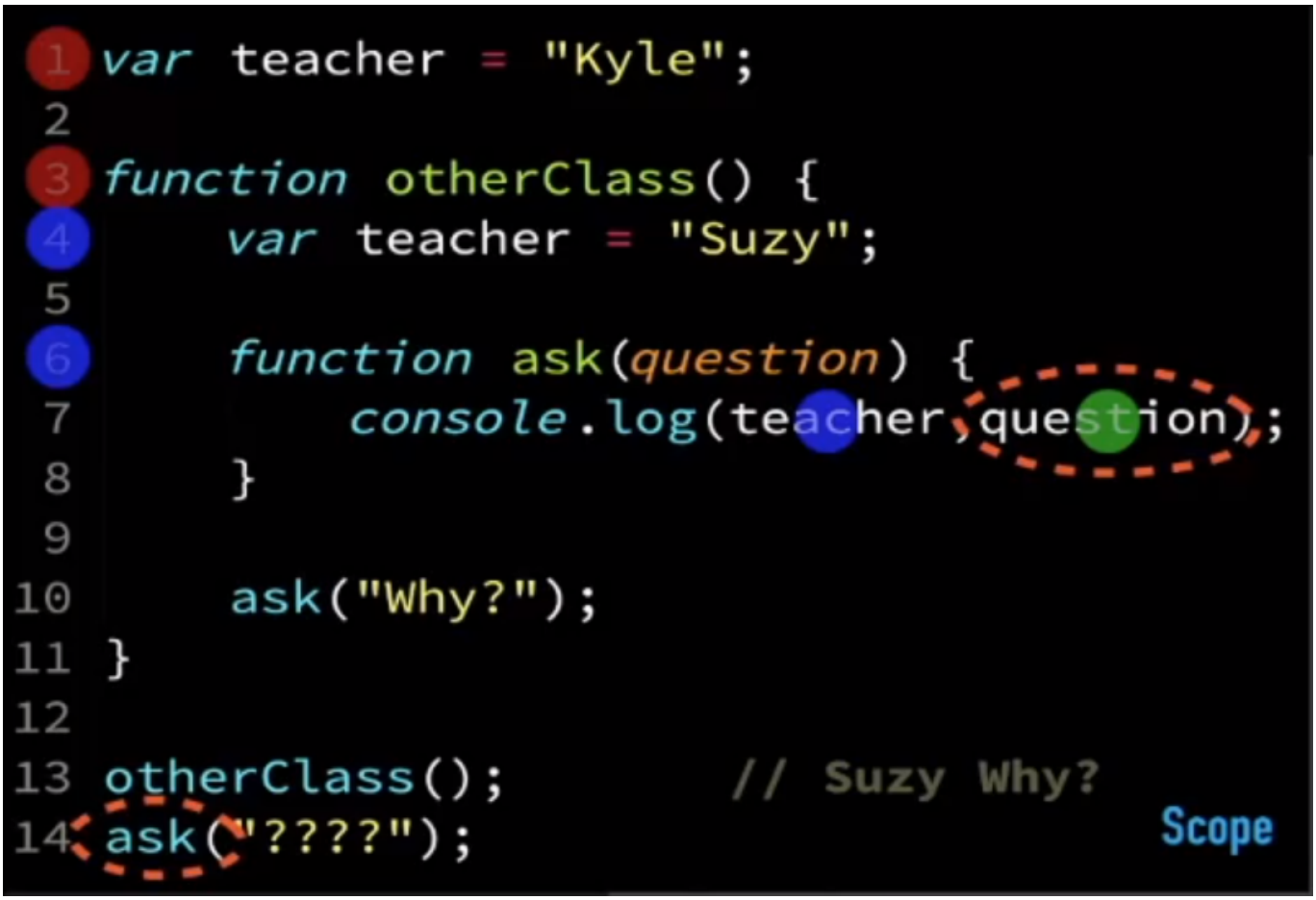
## Nested scopes

```
let teacher = "Kyle"

function otherClass() {
  let teacher = "Suzy";

  function ask(question) {
    console.log(teacher,question);
  }
  ask("Why?");
}

otherClass(); // Suzy Why?
ask("???????");
```

So `let teacher = "Kyle"` creates a red marble (global) `function otherClass() {..}` creates a red marble `function ask(question){..}` creates a blue marble `ask("Why?")` creates a blue marble

`let teacher = "Kyle"` & `function otherClass() {..}` are both declared in global scope which is why they're red marbles

`function ask(question){..}` & `ask("Why?")` are both declared in `otherClass()`'s function scope, which is why they're both blue marbles




In line 14, the invocation of the function `ask()` Our compiler will throw an error: `ReferenceError: ask is not defined`

Because the compiler is going to ask `global scope` if it knows `ask()` And it won't because its nested inside our other function, our compiler has no way of accesing it

---

Function Scopes

## Function Expressions

We've been talking about functions in the compilation phase, adding their identifier as a colored marble in the enclosing scope

```
function teacher () { /* .. */}

var myTeacher = function anotherTeacher() {
  console.log(anotherTeacher);
};

console.log(teacher);
console.log(myTeacher);
console.log(anotherTeacher);
```

`function teacher()` would be a red marble **BUT** line 9 is different from what we've seen

We do know that an identifier called `myTeacher` is going to be created and it's going to be a red marble And we know that there's another function called `anotherTeacher()`, so we need to create a bucket for it at least **But because that function is not a declaration**, we're not going to handle its marble color in the same way The key difference here is that the `anotherTeacher` identifier is going to be a colored marble, **but** at compile time it's gonna be a different colored marble than you expect... not a red marble, but a blue one

```
var myTeacher = function anotherTeacher() {
  console.log(anotherTeacher);
};
```

This is called a **function expression**, as opposed to a **function declaration** A key difference is that, function declarations, add their name (attach their marble) to the enclosing scope, so we make a red marble on line 1, whereas **function expressions** will add their marble to their own scope

But when we run this code we get this: `ReferenceError: anotherTeacher is not defined`

## Named Function Expressions

```
var clickHandler = function() {
  // ..
}

var keyHandler = function keyHandler() {
```

```
  // ..
}
```

How do we know if something's a function declaration? If the word function is literally the first thing in the statement

So if it's not the first thing in the statement, if there's a variable or an operator or parentheses or anything... then it's not a decaraltion, it's an **expression**

`clickHandler` is an anonymous function expression `keyHandler` is a named function expression

3 reasons why you should always prefer named function expressions are:

1. Reliable function self-reference
    - useful if the function is recursive or if the function is an event handler of some sort **and needs to** reference itself to unbind itself
    - It's useful if you need access any properties on that function object such as its length or its name other other thing of that sort
    - Anytime you might need a self reference to the function the single only right answer to that question is, it needs to have a name
2. More debuggable stack traces
    - If you use a named function it shows up in the stack trace, if you use an anonymous function you'll have a harder time debugging
3. More self-documenting code
    - If I have a function that's anonymous, and I look at that function to figure out what that function is doing, I have to read the code body and where it's being passed. I need to infer its purpose from that to understand what the function is doing

## Arrow Functions

```
var ids = people.map(person => person.id); // anonymous functions

var ids = people.map(function getId(person){ // named function
  return persons.id;
})
```

Arrow functions are anonymous functions, and they do have a specific purpose **BUT** they should not be used to replace all functions

### Function Types: Hierarchy

1. (Named) Function Declaration
2. Named Function Expression
3. Anonymous Function Expression

A list from best to worst, and anonymous function can be useful at times, but not should be used more frequently than named functions The point being made is, you want your code to be more readable, not

aesthetically pleasing or cool

## Function Expression Exercise

You are provided three functions stubs `printRecords()` `paidStudentsToEnroll()` `remindUnpaid()`

1. `printRecords()`
    - take a list of student IDs
    - retrieve each student record by its student Id (hint: array `find()`)
    - sort by student name, ascending (hint: array `sort()`)
    - print each record to the console, including `name`,`id`, and `"paid"` or `"not paid"` based on their paid status
2. `paidStudentsToEnroll()`
    - look through all the student records, checking to see which ones are paid but **not yet enrolled**
    - collect these student ids
    - return a new array including the previously enrolled student Ids as well as the to-be-enrolled student Ids (hint: spread `...`)
3. `remindUnpaid()`
    - take a list of student Ids
    - filter this list of student Ids to only those whose records are in unpaid status
    - pass the filtered list to `printRecords()` to print the unpaid reminders

```
function getStudentsById(studentId) {
  return studentRecords.find(function matchId(record){
    return (record.id == studentId)
  })
}

function printRecords(recordIds) {
  var records = recordIds.map(getStudentsById);

  records.sort(function sortByNameAsc(record1,record2){
    if (record1.name < record2.name) {
      return -1;
    }
    else if (record1.name > record2.name) {
      return 1;
    }
    else {
      return 0;
    }
  })

  record.forEach(function printRecord(reord){
    console.log(`${record.name} (${record.id}: ${record.paid} ? "Paid" :
"Not Paid")`);
  })
}

function paidStudentsToEnroll() {
```

```
  let idsToEnroll = studentRecords.filter(function needsToEnroll(record){
    return (record.paid && !currentEnrollment.includes(record.id));
  })
  .map(function getStudentsById(record){
    return record.id;
  })

  return [ ...currentEnrollment, ...idsToEnroll ];
}

function remindUnpaid(recordIds) {
  let unpaidIDs = recordIds.filter(function isUnpaid(studentId){
    let record = getStudentsById(studentId);
    return !record.paid;
  });
  printRecords(unpaidIDs);
}
```

########## Solution but using arrow functions ##########

```
var getStudentsById = studentId =>
  studentRecords.find(
    record => record.id == studentId
  )

var printRecords = recordIds =>
    recordIds.map(getStudentsById)
    .sort(
      (record1,record2) => (record1.name < record2.name) ? -1 :
(record1.name > record2.name) ? 1 : 0
    )
    .forEach(
      record => console.log(`${record.name} (${record.id}: ${record.paid} ?
"Paid" : "Not Paid")`)
    )

var paidStudentsToEnroll = () => [
  ...currentEnrollment,
  ...get(
    studentRecords.filter(
      record => (record.paid && !currentEnrollment.includes(record.id))
    )
    .map(record => record.id)
  )
]

var remindUnpaid = recordIds =>
  printRecords(
    recordIds.filter(
      studentId => !getStudentFromId(studentId).paid
```

```
        )
    )
```