JavaScript Mental Models

```
let a = 10;
let b = a;
a = 0;
```

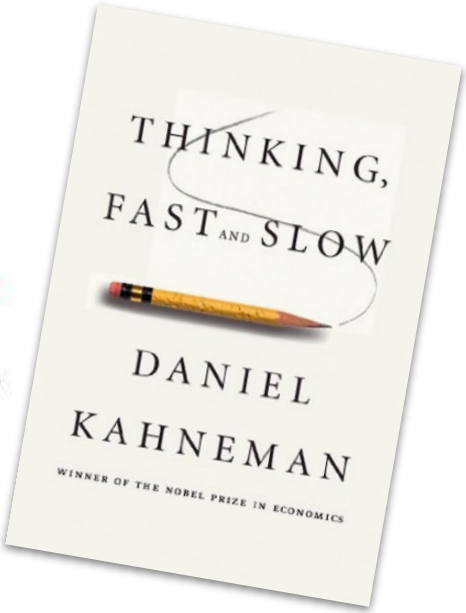
- `let a = 10;`
    - Declare a variable called a. Set it to 10.
- `let b = a;`
    - Declare a variable called b. Set it to a.
    - Wait, what's a again? Ah, it was 10. So b is 10 too.
- `a = 0;`
    - Set the a variable to 0.
- So a is 0 now, and b is 10. That's our answer.

“Thinking, Fast and Slow”

Whenever we can, we rely on the “fast” system.

- Daniel Kahneman


DANIEL KAHNEMAN
AT THE LSE
THINKING, FAST AND SLOW
WE MUST TRUST OUR INSTINCTS
DON'T LIKE THE TERM IRRATIONAL
SYSTEM 1 FAST intuition OPERATES CONTINUSLY
SKILLED BEHAVIORS
ANSWERS
C·O·M·P·L·E·X
QUESTIONS ABSURDLY
PROBLEMS ARISE HERE
SO...
RECOGNIZE WHEN YOU MIGHT MAKE MISTAKES AND CALL UP SYSTEM 2
Wellbeing
SYSTEM 2 SLOW Effortfull MONITORS SYSTEM 1 THOUGHTS
HOW TO MEASURE WELLBEING?
1 HOW SATISFIED ARE YOU?
2 QUALITY OF YOUR EMOTIONAL EXPERIENCE
WHAT DO YOU WANT TO OPTIMIZE?
GOVERNMENTS MUST ASK THEMSELVES
SYSTEM 1 UPDATES CONTINUOUSLY
Cognitive therapy can change reactions
V1
V2
V4
PEOPLE WHO ACCESS Emotion MAKE BETTER DECISIONS
INTERVIEWS HAVE NO CORRELATION TO SUCCESS AT UNIVERSITY
FOR HIGH STAKES DECISIONS WE NEED TO Slow down
SLOW
N.LANE


THINKING, FAST AND SLOW
DANIEL KAHNEMAN
WINNER OF THE NOBEL PRIZE IN ECONOMICS

```
function duplicateSpreadsheet(original) {
  if (original.hasPendingChanges) {
    throw new Error('You need to save the file before you can duplicate
it.');
  }
  let copy = {
    created: Date.now(),
    author: original.author,
    cells: original.cells,
    metadata: original.metadata,
  };
  copy.metadata.title = 'Copy of ' + original.metadata.title;
  return copy;
  }
```

You've probably noticed that:

This function duplicates a spreadsheet.

It throws an error if the original spreadsheet isn't saved.

It prepends "Copy of" to the new spreadsheet's title.

This function also accidentally changes the title of the original spreadsheet.

```
function duplicateSpreadsheet(original) {
  if (original.hasPendingChanges) {
    throw new Error('You need to save the file before you can duplicate
it.');
  }
  let copy = {
    created: Date.now(),
    author: original.author,
    cells: original.cells,
    metadata: original.metadata,
  };
  copy.metadata.title = 'Copy of ' + original.metadata.title;
  return copy;
  }
```

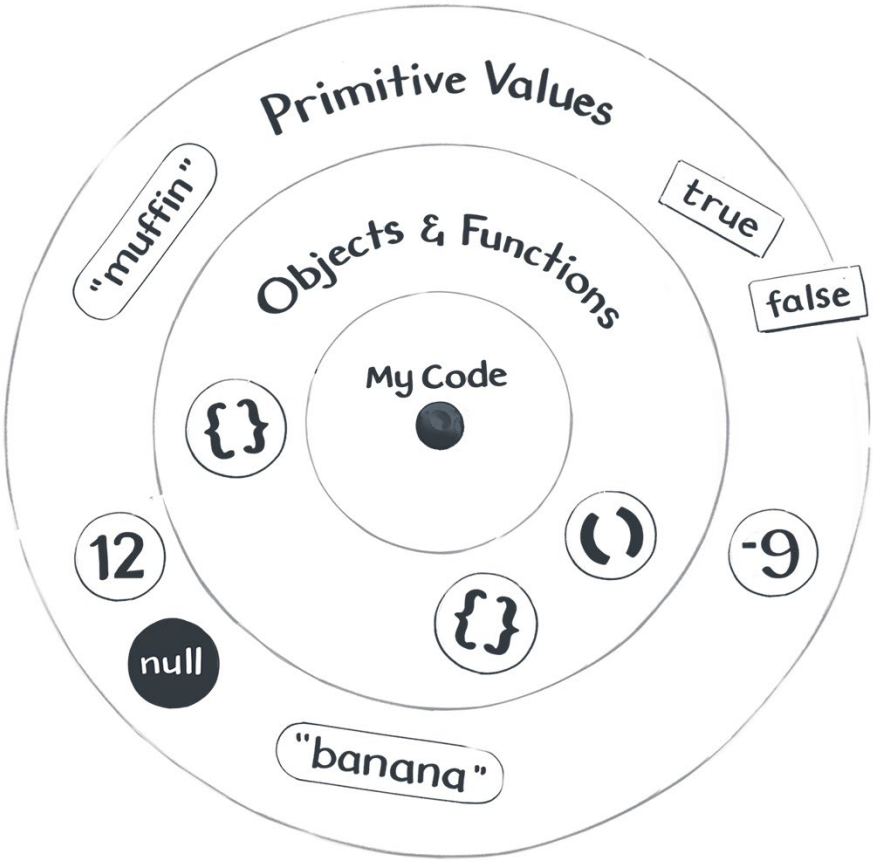
What is a VALUE ?

What is a VALUE ?

This is like asking what a number is in math, or what a point is in geometry.

A value is a ***thing*** in the JavaScript universe.

I'm standing on a small asteroid — it is the code of my program.



But every once in a while, I look up.

Primitive Values

There's nothing I can do in my code that would affect Primitive Values.

```
console.log(2);

console.log("hello");

console.log(undefined);
```

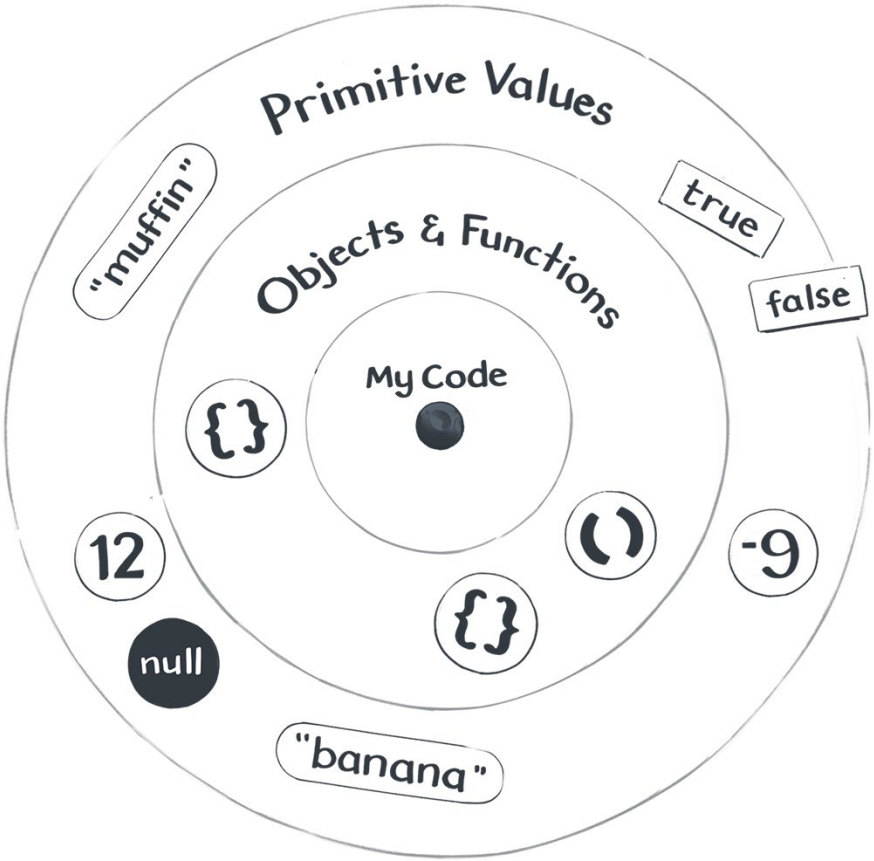
Primitive Values are like stars — **cold and distant, but always there when I need them.**

Objects and Functions

Objects and Functions are also values, but they are not primitive.

They are special because I can *manipulate* them from my code.

```
console.log({});

console.log([]);

console.log(x => x * 2);
```



But every once in a while, I look up.

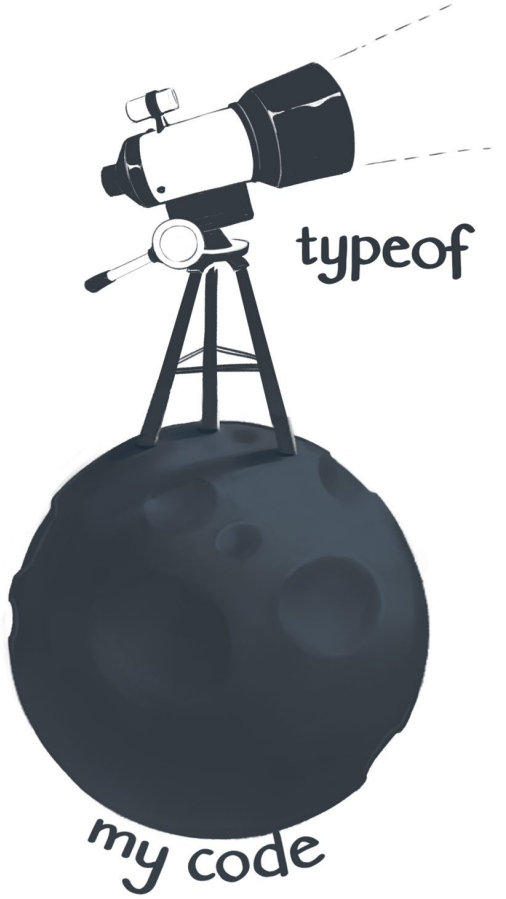
Expressions

Expressions are questions that JavaScript can answer. JavaScript answers expressions in the only way it knows how — with values.

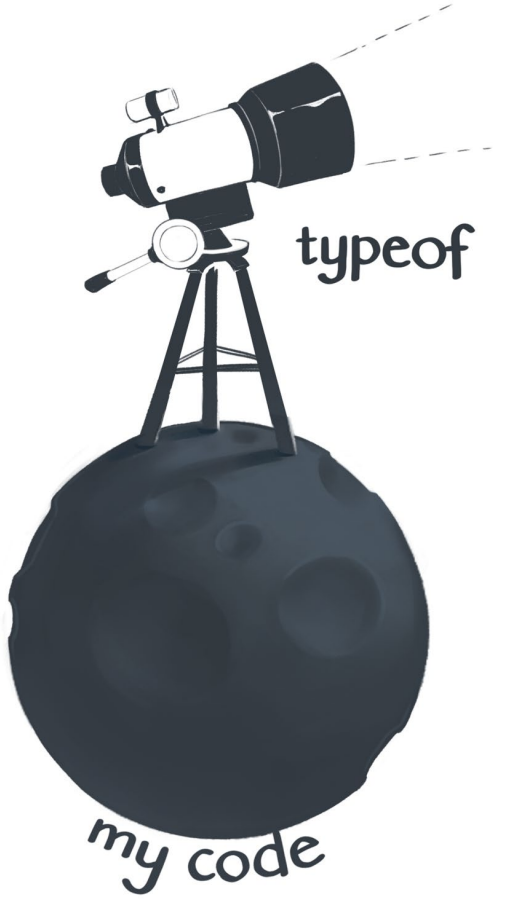
If we “ask” the expression 2 + 2, JavaScript will “answer” with the value 4.

```
console.log(2+2)
```

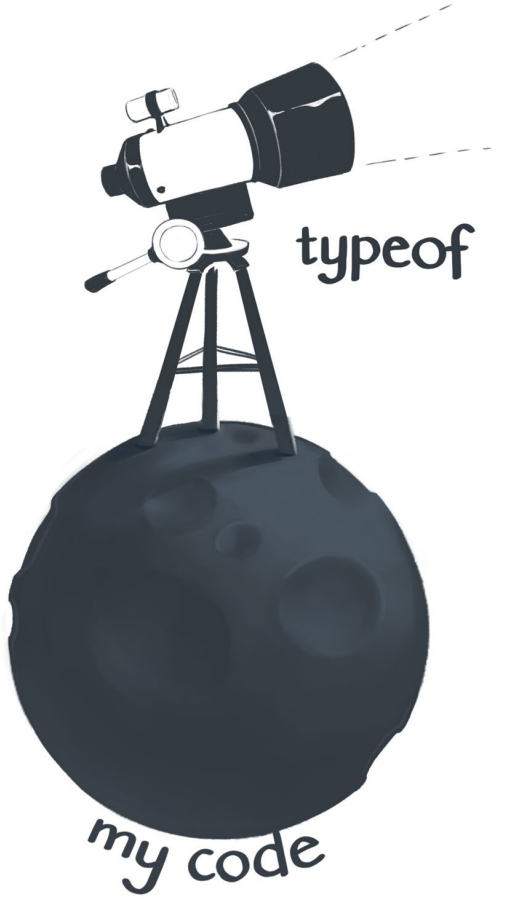
Expressions always result in a **single value**.



Checking a Type




```
// "number"
console.log(typeof(2));


// "string"
console.log(typeof("hello"));


// "undefined"
console.log(typeof(undefined));
```




```
// "object"
console.log(typeof({}));


// "object"
console.log(typeof([]));


// "function"
console.log(typeof(x => x*2));
```

2

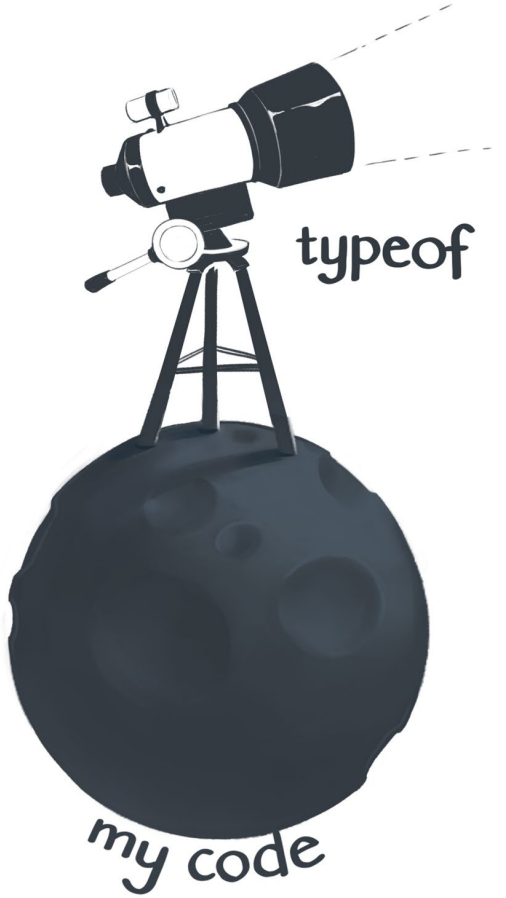
Primitive Values

- **Undefined** (undefined), used for unintentionally missing values.
- **Null** (null), used for intentionally missing values.
- **Booleans** (true and false), used for logical operations.
- **Numbers** (-100, 3.14, and others), used for math calculations.
- **Strings** ("hello", "abracadabra", and others), used for text.
- **Symbols** (uncommon), used to hide implementation details.
- **BigInts** (uncommon and new), used for math on big numbers.

Objects and Functions

- **Objects** ({} and others), used to group related data and code.
- **Functions** (x => x * 2 and others), used to refer to code.

In JavaScript, there are no other fundamental value types other than the ones we have just enumerated.




```
// "object"
console.log(typeof([]));

// "object"
console.log(typeof(new Date()));

// "object"
console.log(typeof(/(hello|goodbye)/));
```

Everything is an object ?

Everything is an object ?

```
"hi".toUpperCase()
```

Primitive Values, such as *numbers* and *strings*, are not objects.

JavaScript creates a wrapper object when you do this, and then immediately discards it.

Thing to Remember

There are values, and then there's everything else.

There are two categories of values:

- Primitive Values
- Objects and Functions.

Some values are lonely:

- `null` is the only value of the Null type
- `undefined` is the only value of the Undefined type

We can ask questions with expressions.

- Interpreted by web browsers

We can inspect the type of something by wrapping it in a typeof expression.

Exercises

Imagine you see some code that checks whether a value is a date:

```
typeof(value) === 'date'
```

Will this code work?

Why or why not?

Exercises

One of the values mentioned in this module “lies” about its type. Concretely, *typeof* will return an incorrect answer for it because of a JavaScript bug that is now too late to fix.

Do you know which value it is? You can **find this value** by trying *typeof* for each example in our list of nine types.

Primitive Values

- **Undefined** (`undefined`), used for unintentionally missing values.
- **Null** (`null`), used for intentionally missing values.
- **Booleans** (`true` and `false`), used for logical operations.
- **Numbers** (`-100`, `3.14`, and others), used for math calculations.
- **Strings** (`"hello"`, `"abracadabra"`, and others), used for text.
- **Symbols** (uncommon), used to hide implementation details.
- **BigInts** (uncommon and new), used for math on big numbers.

Objects and Functions

- **Objects** (`{}` and others), used to group related data and code.
- **Functions** (`x => x * 2` and others), used to refer to code.

Exercises

No matter what value we pick, we know that typeof(value) can only give us one of the several predetermined answers.

Is there anything similarly interesting that we can say about typeof(typeof(value))?

Explain your thinking process.

Exercises

```
let reaction = 'yikes';
reaction[0] = 'l';
console.log(reaction);
```

Primitive Values Are Immutable

```
let arr = [212, 8, 506];
let str = 'hello';
```

You can access the first array item similarly to how you would access a string's first character. It almost feels like strings are arrays (but they're not!):

An empty string is a string, too:

```
console.log(typeof('')); // "string"
```

```
console.log(arr[0]); // 212
console.log(str[0]); // "h"
```

```
arr[0] = 420;
console.log(arr); // [420, 8, 506]
```

```
str[0] = 'j'; // ???
```

All primitive values are immutable.

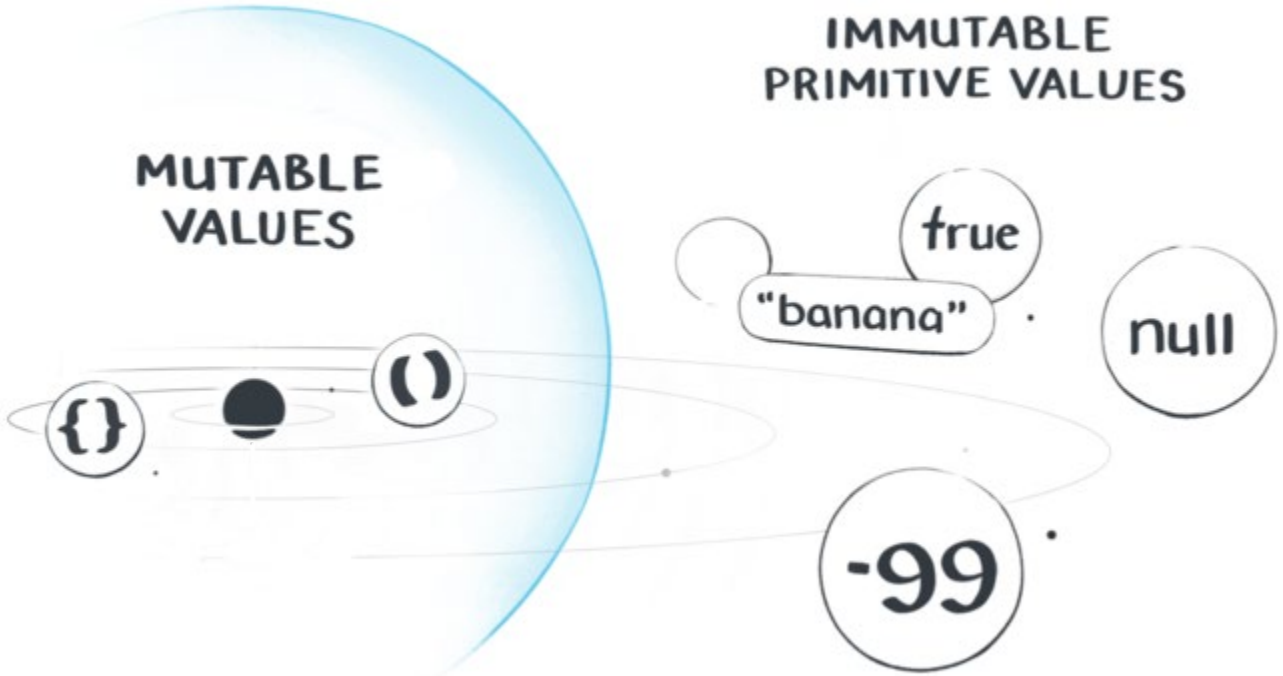
“Immutable” is a fancy Latin way to say “unchangeable”.

Read-only.

You can’t mess with primitive values. At all.

```
let fifty = 50;
fifty.shades = 'gray'; // No!
```

U Can't Touch This


MUTABLE
VALUES
{}
()
IMMUTABLE
PRIMITIVE VALUES
true
"banana"
null
-99

```
let pet = 'Narwhal';

pet = 'The Kraken';

console.log(pet); // ?
```

Is this a contradiction?

Variables are not values

Variables are Wires

Variables point to values.

```
let pet = 'Narwhal'
```

For example, I can point the pet variable at the "Narwhal" value:

```
let pet = 'Narwhal';
```

Assigning a Value to a Variable


```
let pet = 'Narwhal'
pet = 'The Kraken'
```

I can assign some other value to my variable:

```
pet = 'The Kraken';
```

Try it in the console.

```
'war' = 'peace';
```

The left side of an assignment must be a “wire”.

The right side of an assignment must be an expression.

The right side of an assignment must be an expression.

```
pet = count + ' Dalmatians';
```

Here, count + ' Dalmatians' is an expression — a question to JavaScript.

JavaScript will answer it with a value (for example, "101 Dalmatians").

From now on, the pet “wire” will start pointing to that value.

Numbers like `2` or strings like `'The Kraken'` written in code are also expressions?

Yes! Such expressions are called *literals* — because we literally write down their values.

What is the current value of pet?

```
console.log(pet);
```

Reading a Value of a Variable

We can't really pass *variables* to functions.

We pass *the current value* of the pet variable.

A variable name serves as an expression too!

A variable name serves as an expression too!

The same expression can give us different values at different times!

Example

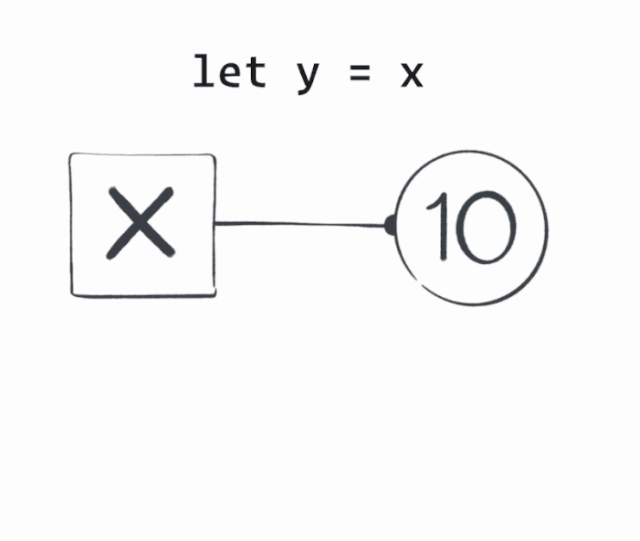
```
function double(x) {

  x = x * 2;

}

let money = 10;

double(money);

console.log(money); // ?
```

```
let a = 10;
let b = a;
a = 0;
```

```
let x = 10
```

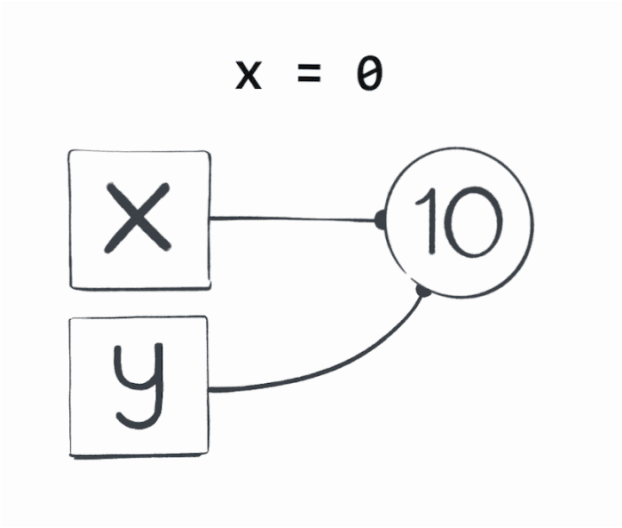
- Declare a variable called x.
  - *Make a wire for the x variable.*

- Assign to x the value of 10.
  - ***Point x's wire to the value 10.***



- Declare a variable called *y*.
  - *Make a wire for the y variable.*

- Assign to *y* the value of `x`.
  - Evaluate the expression: `x`.
    i. *The "question" we want to answer is* `x`.
    ii. ***Follow the x's wire — the answer is the value 10.***
  - The `x` expression resulted in the value `10`.
  - Therefore, assign to *y* the value of `10`.
  - ***Point y's wire to the value 10.***



Assign to x the value of 0.

- ***Point x's wire to the value 0.***

Variables always point at *values*.

Thing to Remember

Primitive values are immutable.

Variables are not values.

- Each variable points to a particular value.
- We can change which value it points to by using the = assignment operator.

Variables are like wires.

- A “wire” is not a JavaScript concept — but it helps us imagine how variables point to values.

Look out for contradictions.

Nouns and verbs matter.

Exercises

What happens if we run this code?

Is this code valid? Why or why not?

```
let numberOfTentacles = 10;
numberOfTentacles = 'eight';
console.log(typeof(numberOfTentacles));
```

Exercises

Here's a slightly different example. What happens if we run it?

Is there a difference in behavior compared to the last snippet? Explain why or why not using our mental model.

```
let numberOfTentacles = 10;
console.log(typeof(numberOfTentacles));
numberOfTentacles = 'eight';
```

Exercises

What happens if we run this code?

Explain why using our mental model.

```
let answer = true;
answer.opposite = false;
console.log(answer.opposite);
```

Exercises

What happens if we run this code?

Explain why using our mental model.

```
null = 10;
console.log(null);
```

Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

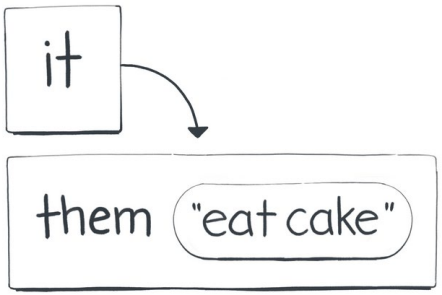
```
let it = 'be';
let them = 'eat cake';
it = them;
```

Exercises

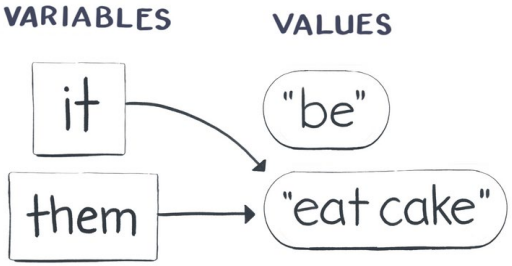
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let it = 'be';
let them = 'eat cake';
it = them;
```

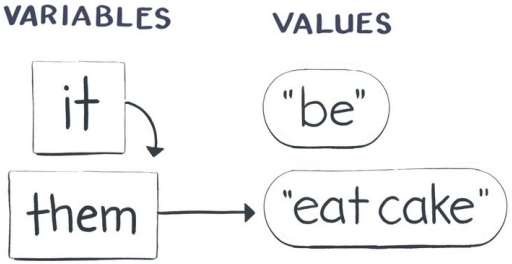
A)



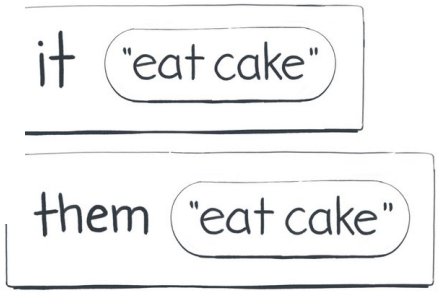
B)



C)



D)



Exercises

This code prints "*T*" to the console. The *feed* function is written by our colleague **in another file**. We don't know what it does.

Can our colleague change the console output only by editing the **feed** function? Why or why not?

```
let pets = 'Tom and Jerry';
feed(pets);
console.log(pets[0]);
```

Exercises

This code prints "*Tom*" to the console. The *feed* function is written by our colleague **in another file**. We don't know what it does.

Can our colleague change the console output only by editing the *feed* function? Why or why not?

```
let pets = ['Tom', 'and', 'Jerry'];
feed(pets);
console.log(pets[0]);
```

Primitive Values

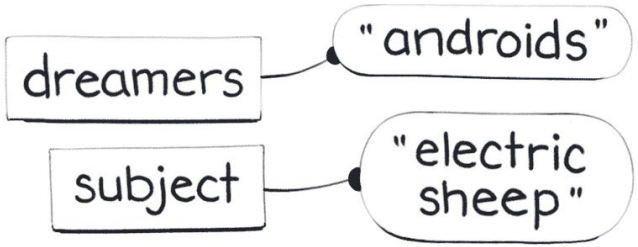
- **Undefined** (`undefined`), used for unintentionally missing values.
- **Null** (`null`), used for intentionally missing values.
- **Booleans** (`true` and `false`), used for logical operations.
- **Numbers** (`-100`, `3.14`, and others), used for math calculations.
- **Strings** (`"hello"`, `"abracadabra"`, and others), used for text.
- **Symbols** (uncommon), used to hide implementation details.
- **BigInts** (uncommon and new), used for math on big numbers.

Objects and Functions

- **Objects** (`{}` and others), used to group related data and code.
- **Functions** (`x => x * 2` and others), used to refer to code.

Anything not on this list is an object.

The JavaScript Simulation



Studying From the Outside

A string of text is a sequence of bytes stored inside a silicon chip.

Mental focus on the physical world of people and computers.

Studying From the Inside

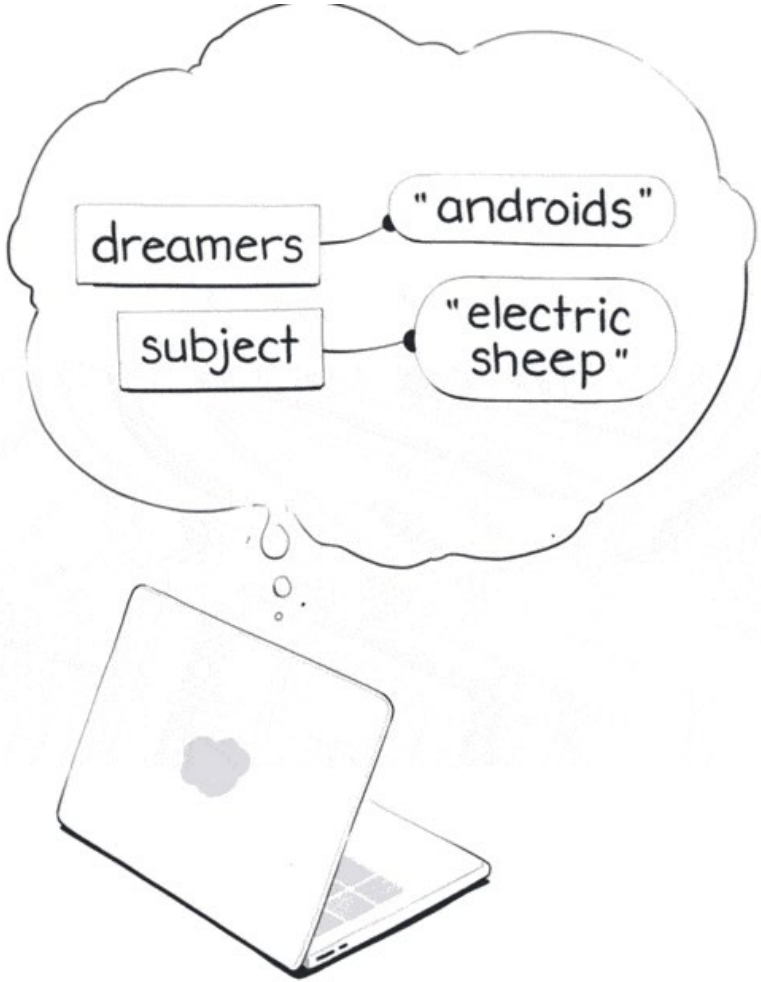
A string of text is a *value.*

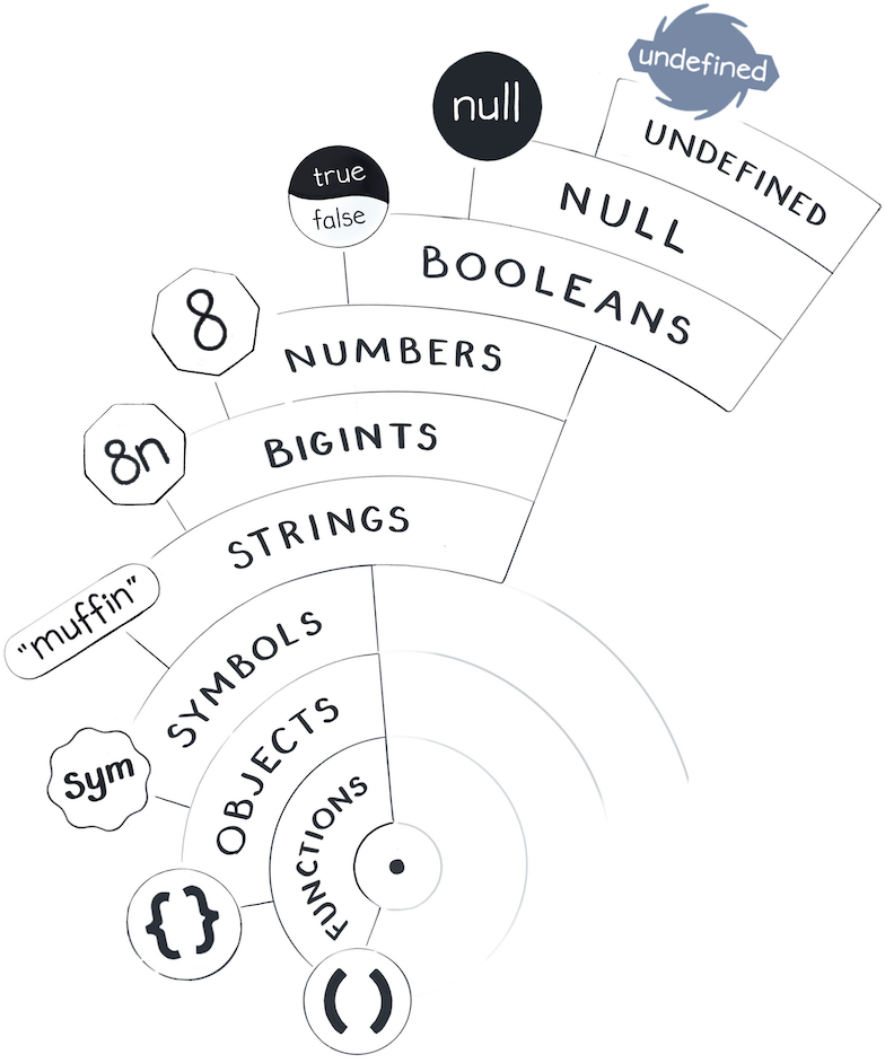
Mental focus on the JavaScript world for what it is — without thinking about how it's implemented.

Each value belongs to one of a few built-in types.

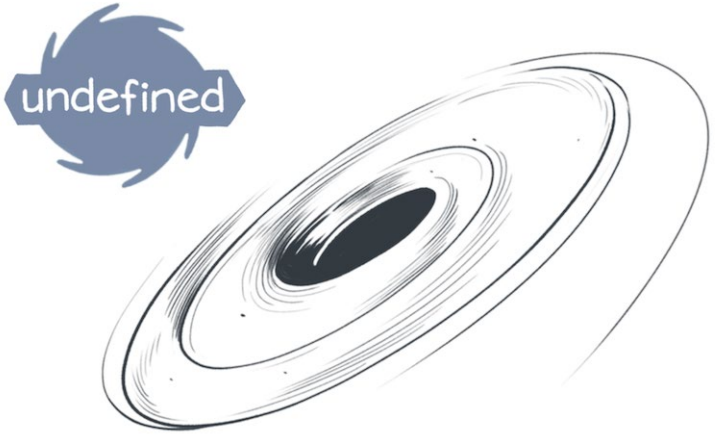
Some of them are primitive, which makes values of those types immutable.

Variables are “wires” pointing from names in our code to values.


dreamers
"androids"
subject
"electric sheep"



Counting the Values



Undefined

There is only one value of that type — `undefined`.

```
// "undefined"
console.log(typeof(undefined));
```

```
let person = undefined;
console.log(person.mood); // ?
```

```
let person = undefined;
console.log(person.mood); // TypeError!
```

Reading a property from undefined will break your program.

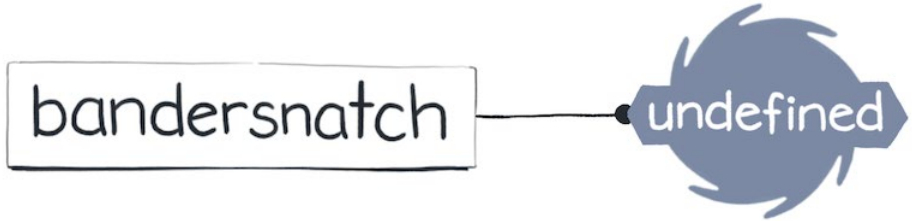
In JavaScript, `undefined` represents the concept of an *unintentionally missing value*.

`undefined` also “occurs naturally”.

It shows up in some situations where JavaScript doesn’t know what value you wanted.

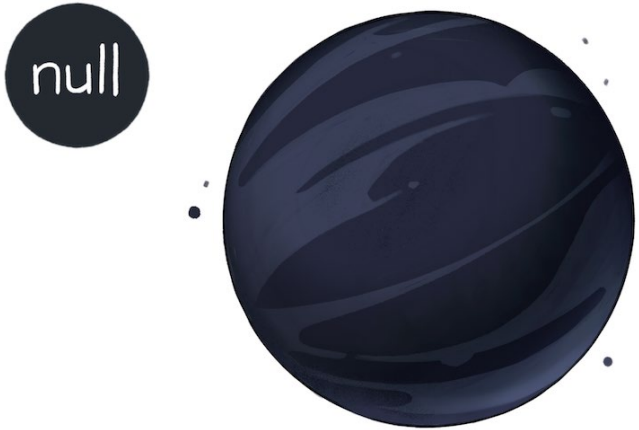
undefined also “occurs naturally”.

```
let bandersnatch;
console.log(bandersnatch); // undefined
```



```
console.log(jabberwocky); // ?
let jabberwocky;
```

```
console.log(jabberwocky); // ReferenceError!
let jabberwocky;
```



Null

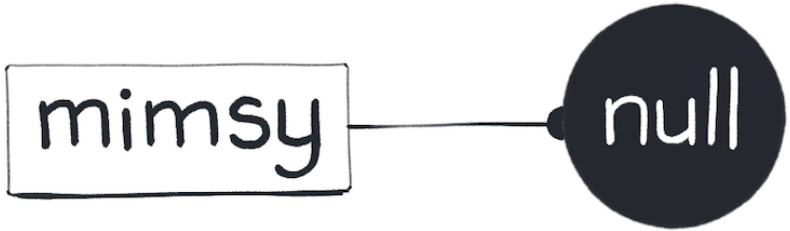
`null` is the only value of its own type.

```
// "object"
console.log(typeof(null));
```

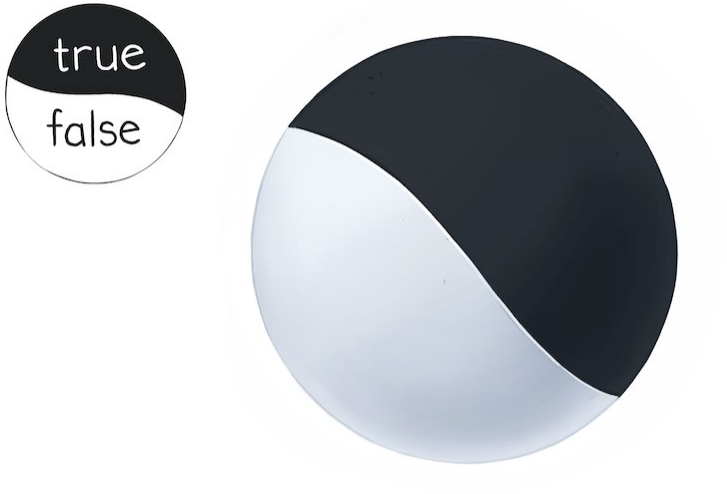
```
let mimsy = null;
console.log(mimsy.mood); // ?
```

Null

```
let mimsy = null;
console.log(mimsy.mood); // TypeError!
```



In JavaScript, `null` is used for *intentionally missing values*.



Booleans

There are only two boolean values: **`true`** and **`false`**.

```
// "boolean"
console.log(typeof(true));

// "boolean"
console.log(typeof(false));
```

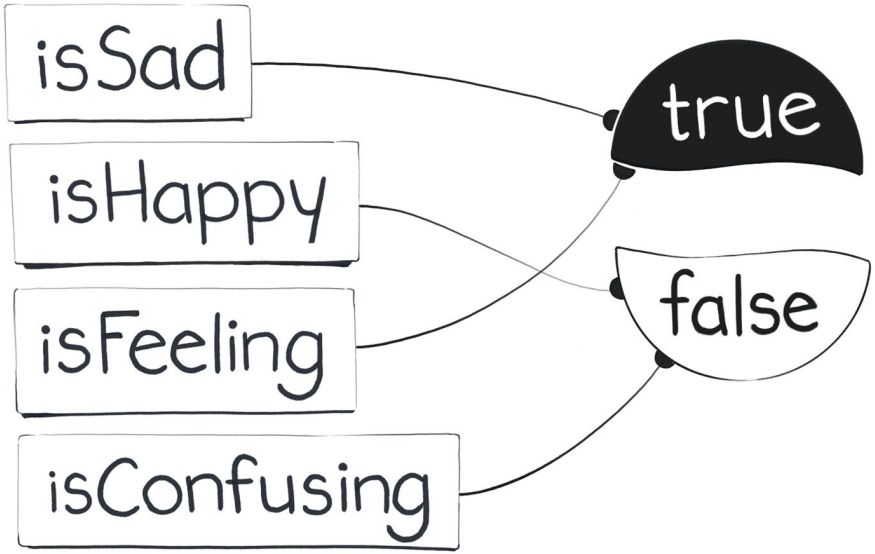
We can perform logical operations with Booleans

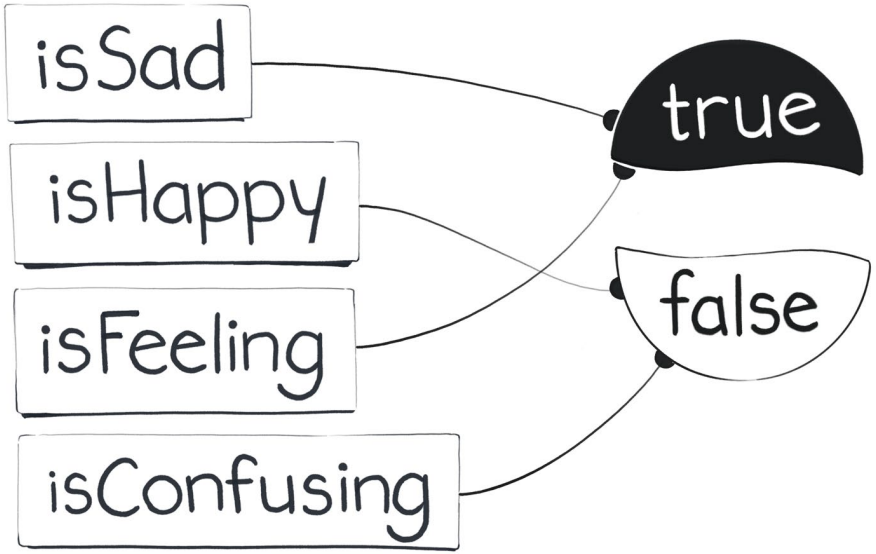
```
let isSad = true;
let isHappy = !isSad; // The opposite
let isFeeling = isSad || isHappy; // Is at least one of them true?
let isConfusing = isSad && isHappy; // Are both true?
```

Exercises

Sketch out the variables, the values, and the wires between them for the above snippet of code.

```
let isSad = true;
let isHappy = !isSad; // The opposite
let isFeeling = isSad || isHappy; // Is at least one of them true?
let isConfusing = isSad && isHappy; // Are both true?
```


isSad
isHappy
isFeeling
isConfusing
true
false



There is only one **`true`** and one **`false`** value.

Numbers

```
// "number"
console.log(typeof(28));

// "number"
console.log(typeof(3,14));

// "number"
console.log(typeof(-140));
```

There are only eighteen quintillion, four hundred and thirty-seven quadrillion, seven hundred and thirty-six trillion, eight hundred and seventy-four billion, four hundred and fifty-four million, eight hundred and twelve thousand, six hundred and twenty-four **values.**

JavaScript numbers don’t behave exactly the same way as regular mathematical numbers do.

```
console.log(0.1 + 0.2 === 0.3); // ?

console.log(0.1 + 0.2 === 0.30000000000000004); // ?
```

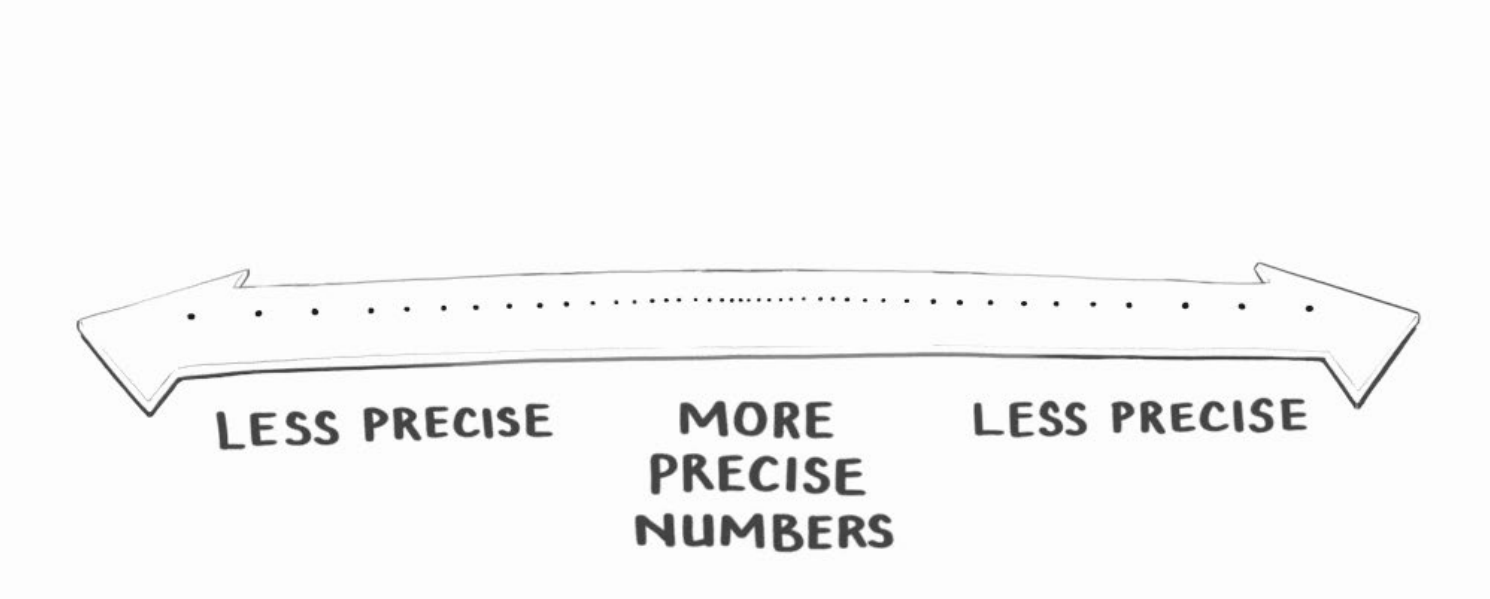
```
console.log(0.1 + 0.2 === 0.3); // false

console.log(0.1 + 0.2 === 0.30000000000000004); // true
```

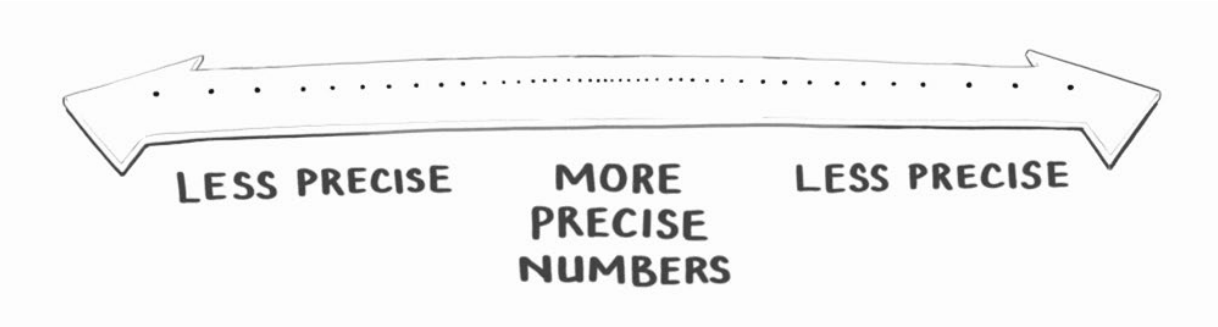
Floating point math — JavaScript treats numbers as having a limited precision.

Have you ever used a scanner to turn a physical photo or a document into a digital one?

This analogy can help us understanding JavaScript numbers.

Floating Point Math






```
console.log(Number.MAX_SAFE_INTEGER);     // 9007199254740991
console.log(Number.MAX_SAFE_INTEGER + 1); // 9007199254740992
console.log(Number.MAX_SAFE_INTEGER + 2); // 9007199254740992
console.log(Number.MAX_SAFE_INTEGER + 3); // 9007199254740994
console.log(Number.MAX_SAFE_INTEGER + 4); // 9007199254740996
console.log(Number.MAX_SAFE_INTEGER + 5); // 9007199254740996
```

Any numbers between Number.MIN_SAFE_INTEGER and Number.MAX_SAFE_INTEGER are exact.

This is why `10 + 20 === 30`

Learn more about floating point math

Special Numbers

```
let scale = 0;
let a = 1 / scale; // Infinity
let b = 0 / scale; // NaN
let c = -a; // -Infinity
let d = 1 / c; // -0
```

```
console.log(typeof(NaN)); // "number"
```

`NaN`, which is the result of `0/0` and some other invalid math, stands for “not a number”.

`NaN` is called *"not a number"* because it represents an invalid result in the floating point math.

- **Undefined**: Only one value, undefined.
- **Null**: Only one value, null.
- **Booleans**: Two values: true and false.
- **Numbers**: One value for each floating point math number.

Thing to Remember

Not all numbers can be perfectly represented in JavaScript.

- Their decimal point is “floating”.

Numbers from invalid math operations like `1/0` or `0/0` are special.

`typeof(NaN)` is a number because `NaN` is a numeric value.

Exercises

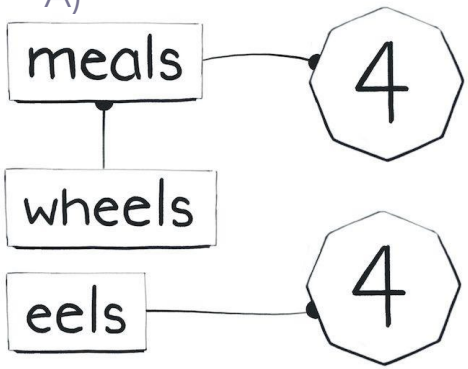
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

Exercises

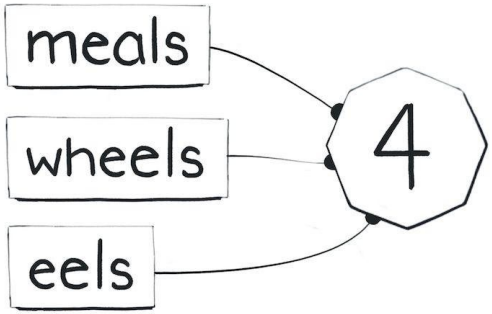
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let meals = 4;
let wheels = meals;
let eels = 2 + 2;
```

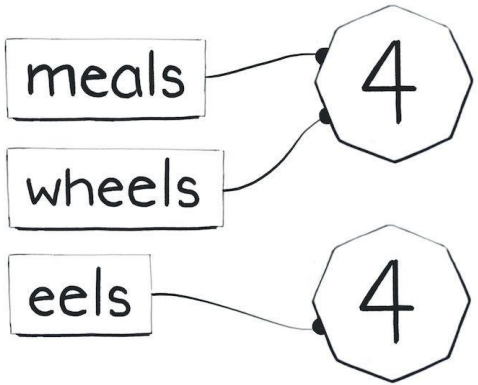
A)



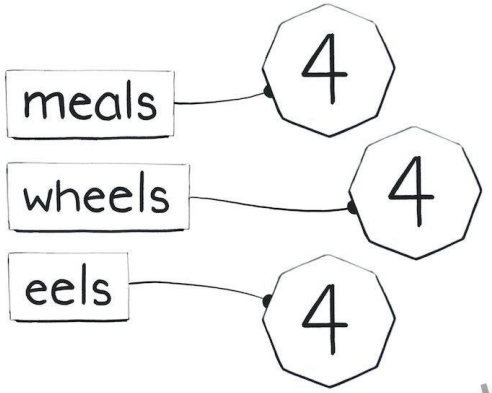
B)



C)



D)



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

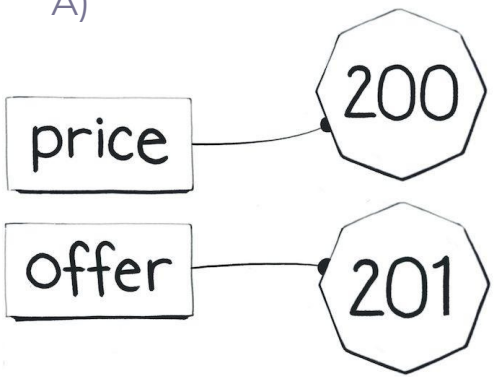
```
let price = 100;
let offer = price + 1;
price = 200;
```

Exercises

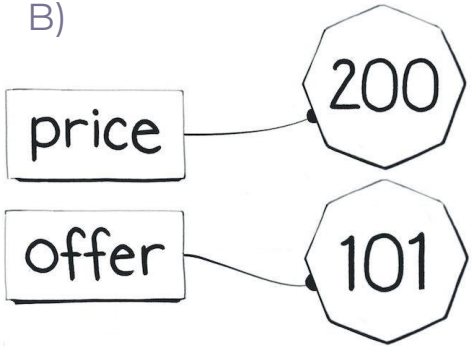
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let price = 100;
let offer = price + 1;
price = 200;
```

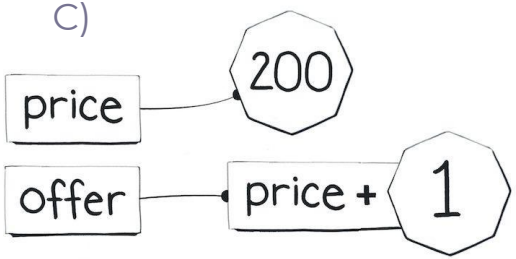
A)



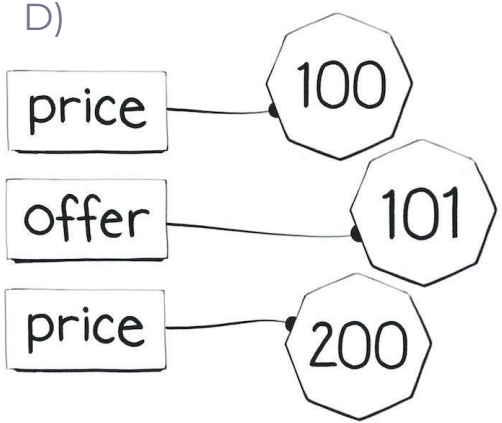
B)



C)



D)



Exercises

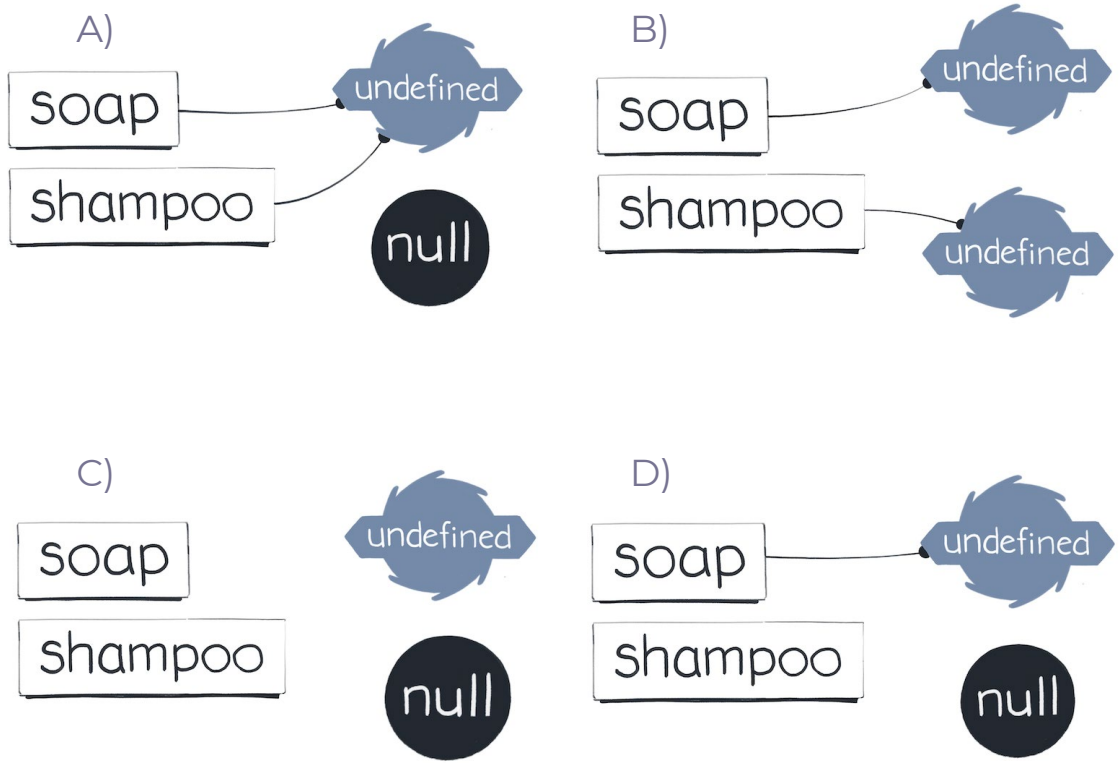
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

```
let shampoo;
let soap = null;
soap = shampoo;
```

Exercises

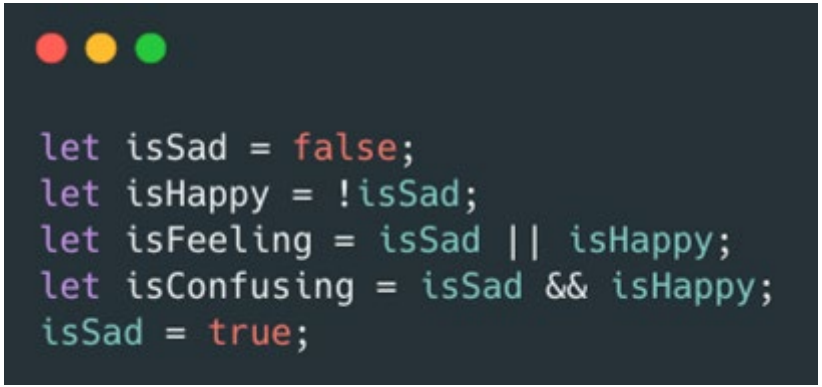
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let shampoo;
let soap = null;
soap = shampoo;
```



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.



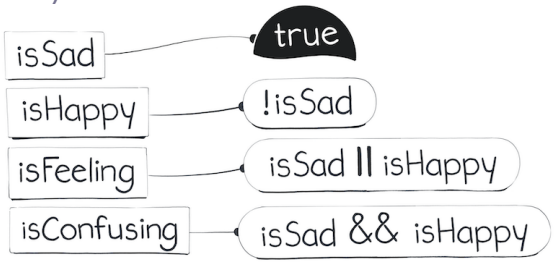
```
let isSad = false;
let isHappy = !isSad;
let isFeeling = isSad || isHappy;
let isConfusing = isSad && isHappy;
isSad = true;
```

Exercises

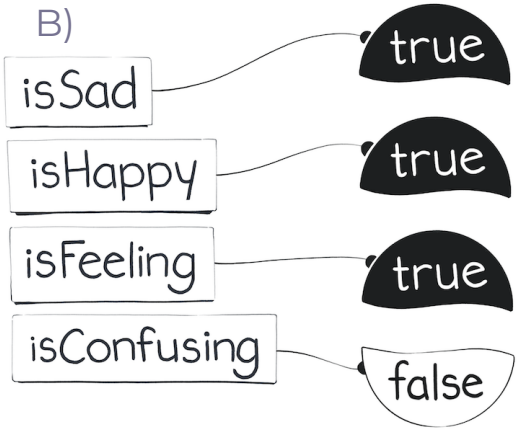
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let isSad = false;
let isHappy = !isSad;
let isFeeling = isSad || isHappy;
let isConfusing = isSad && isHappy;
isSad = true;
```

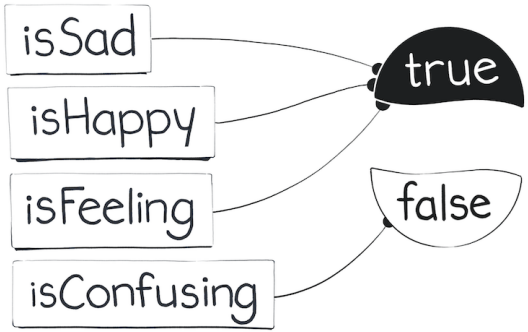
A)



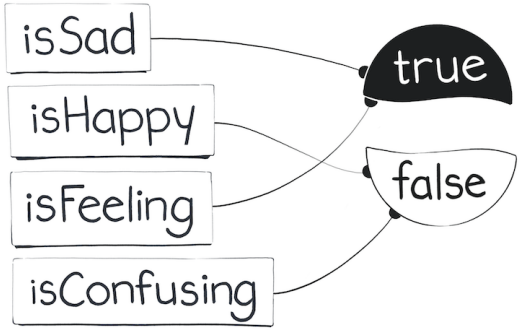
B)

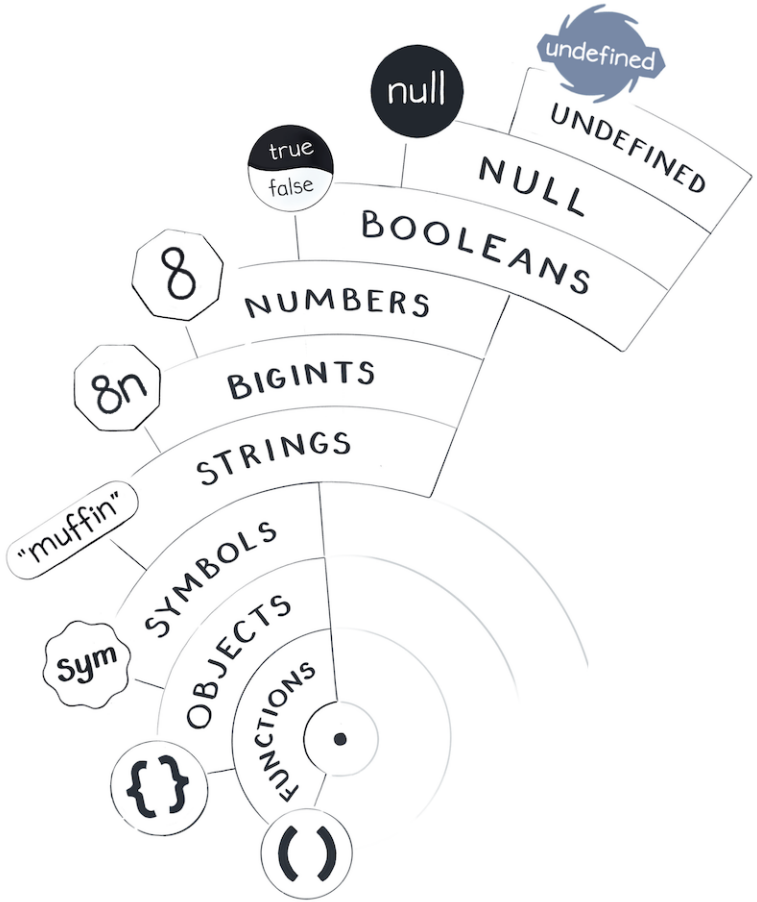


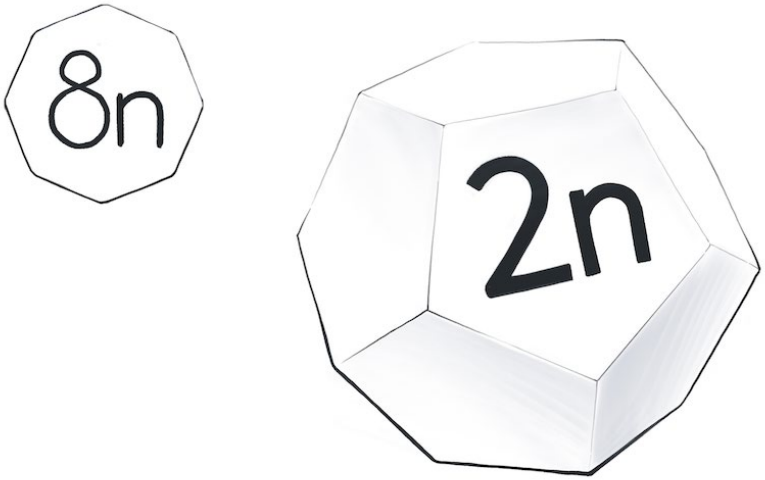
C)



D)




undefined
null
UNDEFINED
true
false
NULL
BOOLEANS
8
NUMBERS
8n
BIGINTS
STRINGS
"muffin"
SYMBOLS
Sym
OBJECTS
{}
FUNCTIONS
( )



BigInts

There is an infinite number of BigInts.

BigInts represent large integers with precision.

```
let alot = 9007199254740991n; // Notice n at the end
```

BigInts

```
console.log(alot + 1n); // 9007199254740992n
console.log(alot + 2n); // 9007199254740993n
console.log(alot + 3n); // 9007199254740994n
console.log(alot + 4n); // 9007199254740995n
console.log(alot + 5n); // 9007199254740996n
```

Strings

Strings represent text in JavaScript.

There are three ways to write strings (single quotes, double quotes, and backticks)

```
// "string"
console.log(typeof("こんにちは"));

// "string"
console.log(typeof('こんにちは'));

// "string"
console.log(typeof(`こんにちは`));
```

An empty string is a string, too:

```
console.log(typeof('')); // "string"
```

Strings Aren't Objects

All strings have a few built-in properties.

```
let cat = 'Cheshire';

console.log(cat.length); // 8

console.log(cat[0]); // "C"

console.log(cat[1]); // "h"
```

A Value for Every Conceivable String

In JavaScript, there is a distinct value for every conceivable string

This includes:

- Your grandmother's maiden name
- The fanfic you published ten years ago under an alias
- The script of Matrix 5 which hasn't been written yet

Does this code create a string?

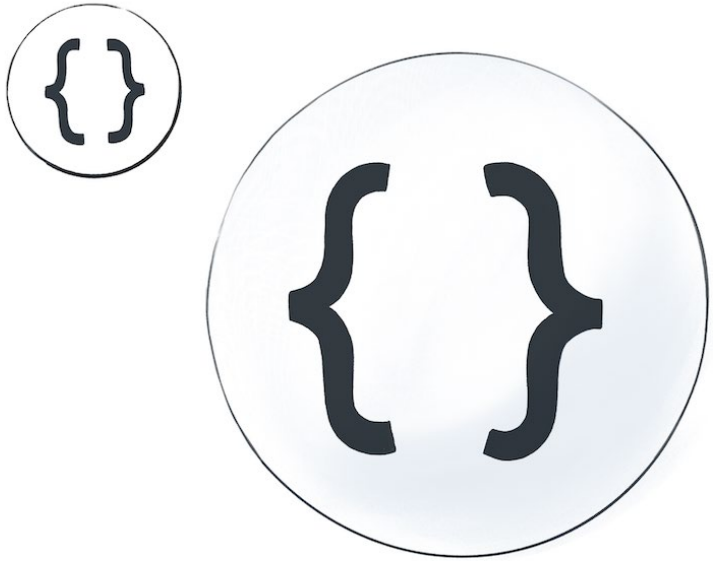
```
// Try it in your console
let answer = prompt('Enter your name');
console.log(answer); // ?
```

To keep our mental model simple, we will say that **all conceivable string values already exist from the beginning** — one value for every distinct string.

Symbols

Symbols are a relatively recent addition to the language.

```
let alohomora = Symbol();
console.log(typeof(alohomora)); // "symbol"
```



Objects

Objects include arrays, dates, RegExps, and other non-primitive values:

```
console.log(typeof({})); // "object"
console.log(typeof([])); // "object"
console.log(typeof(new Date())); // "object"
console.log(typeof(/\d+/)); // "object"
console.log(typeof(Math)); // "object"
```

Object properties can be accessed with . or [ ]:

```
let rapper = { name: 'Malicious' };
rapper.name = 'Malice'; // Dot notation
rapper['name'] = 'No Malice'; // Bracket notation
```

Objects

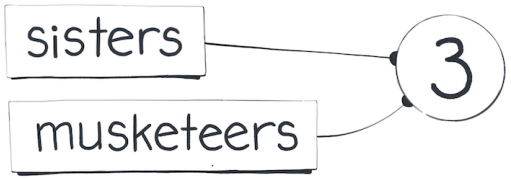
Objects are not primitive values.

By default, objects are mutable.

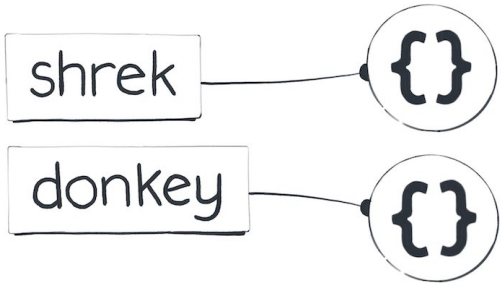
```
for (let i = 0; i < 7; i++) {
  console.log( function() {} );
}
```

Making Our Own Objects

```
let sisters = 3;

let musketeers = 3;
```




```
let shrek = {};

let donkey = {};
```



Every time we use the **{}** object literal, we ***create*** a brand new object value

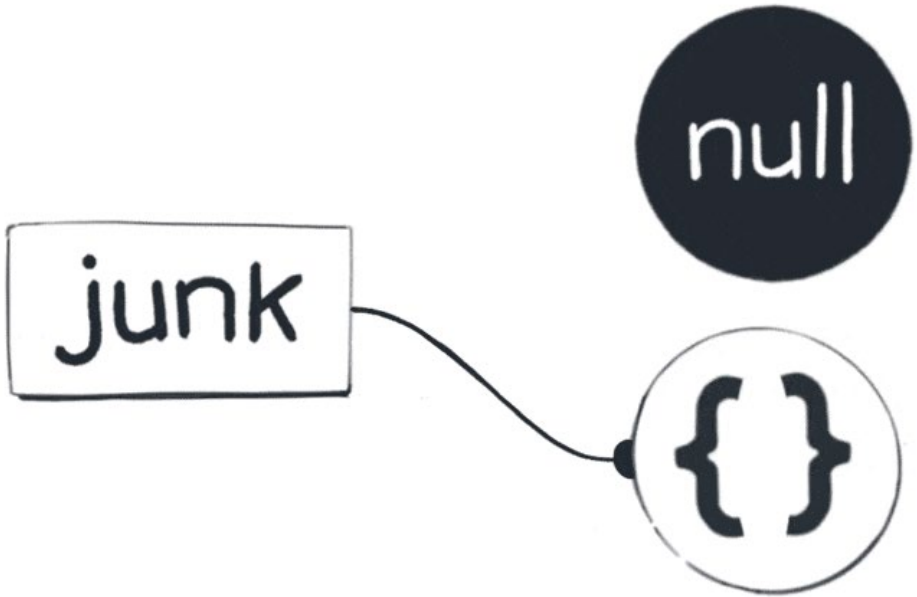
Do Objects Disappear?

Do Objects Disappear?

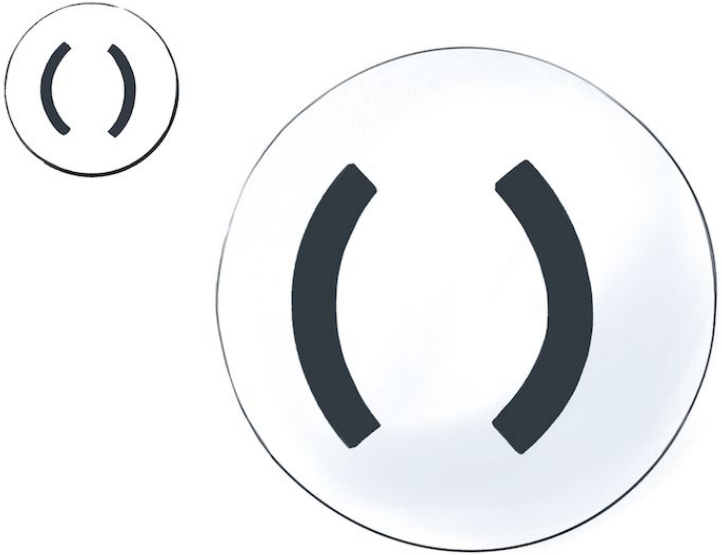
JavaScript is designed in a way that we can’t tell one way or the other from inside our code.

```
let junk = {};

junk = null; // Doesn't necessarily destroy an object
```

JavaScript is a garbage-collected language.



Although we can’t destroy an object, it might eventually “disappear” if there is no way to reach it.



Functions

A function is a JavaScript procedure — a set of statements that performs a task or calculates a value.

Functions Are Values.

To use a function, you must define it somewhere in the scope from which you wish to call it.

How many different values does it pass to console.log?

```
for (let i = 0; i < 7; i++) {
  console.log(2);
}
```

How many different values does it pass to console.log?

```
for (let i = 0; i < 7; i++) {
  console.log({});
}
```

How many different values does it pass to console.log?

```
for (let i = 0; i < 7; i++) {
  console.log(function() {});
}
```

Calling a Function

What does this code print?

```
let countDwarves = function()
{ return 7; };

let dwarves = countDwarves;

console.log(dwarves);
```

1. First, we created a new function value with a **`function() { }`** expression, and pointed the **`countDwarves`** variable at this value.
2. Next, we pointed the dwarves variable at the value that **`countDwarves`** is pointing to — which is the same function value.
3. Finally, we logged the value that dwarves is currently pointing to.

```
let countDwarves = function()
{ return 7; };

let dwarves = countDwarves;

console.log(dwarves);
```

Calling a Function

What does this code print?

```
let countDwarves = function()
{ return 7; };

// () is a function call
let dwarves = countDwarves();

console.log(dwarves);
```

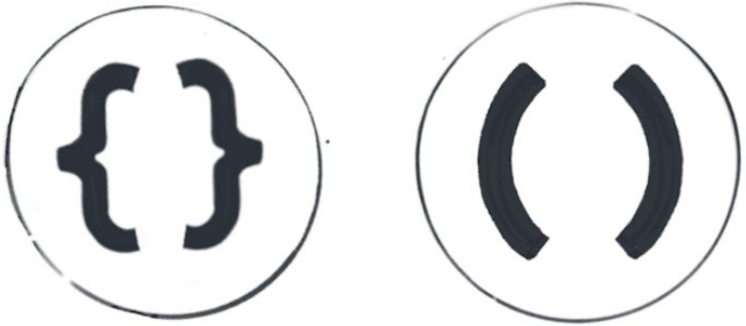
- **`let dwarves = countDwarves`** means “Point dwarves towards the value that **`countDwarves`** is pointing to.”
- **`let dwarves = countDwarves()`** means “Point dwarves towards the value **returned by** the function that **`countDwarves`** is pointing to.”

```
let countDwarves = function()
{ return 7; };

// () is a function call
let dwarves = countDwarves();

console.log(dwarves);
```

`countDwarves()` is an expression.

To “answer” a call expression,
JavaScript runs the code inside our function,
and hands us the returned value as the result.

- **Undefined**: Only one value, undefined.
- **Null**: Only one value, null.
- **Booleans**: Two values: true and false.
- **Numbers**: One value for each floating point math number.
- **BigInts**: One value for every conceivable integer.
- **Strings**: One value for every conceivable string.
- **Symbols**: We skipped Symbols for now, but we'll get to them someday!



The types below are special because they let us *make our own values*:

- **Objects**: One value for every object literal we execute.
- **Function**: One value for every function expression we execute.

Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

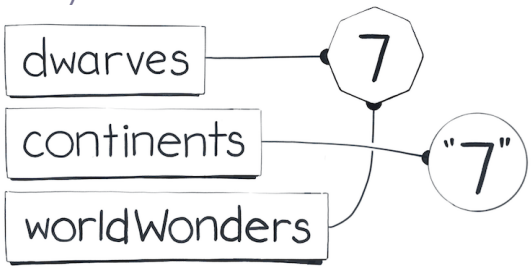
```
let dwarves = 7;
let continents = '7';
let worldWonders = 3 + 4;
```

Exercises

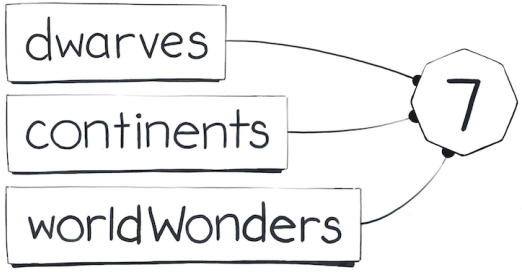
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let dwarves = 7;
let continents = '7';
let worldWonders = 3 + 4;
```

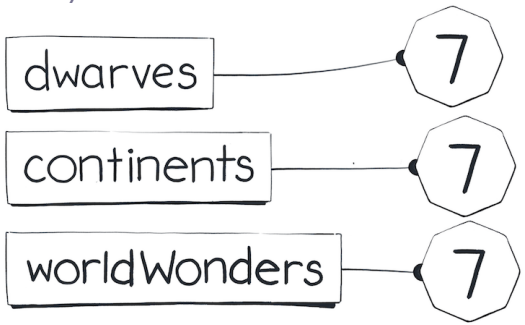
A)



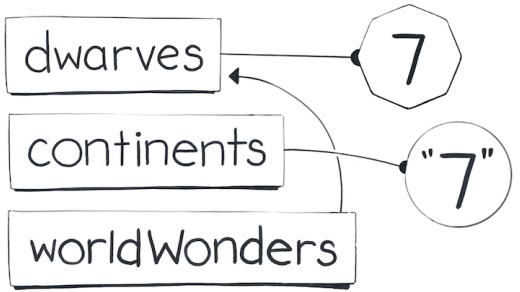
B)



C)

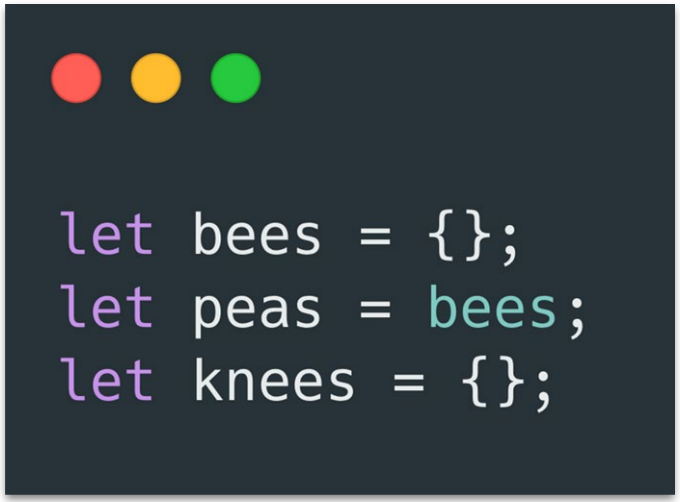


D)



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

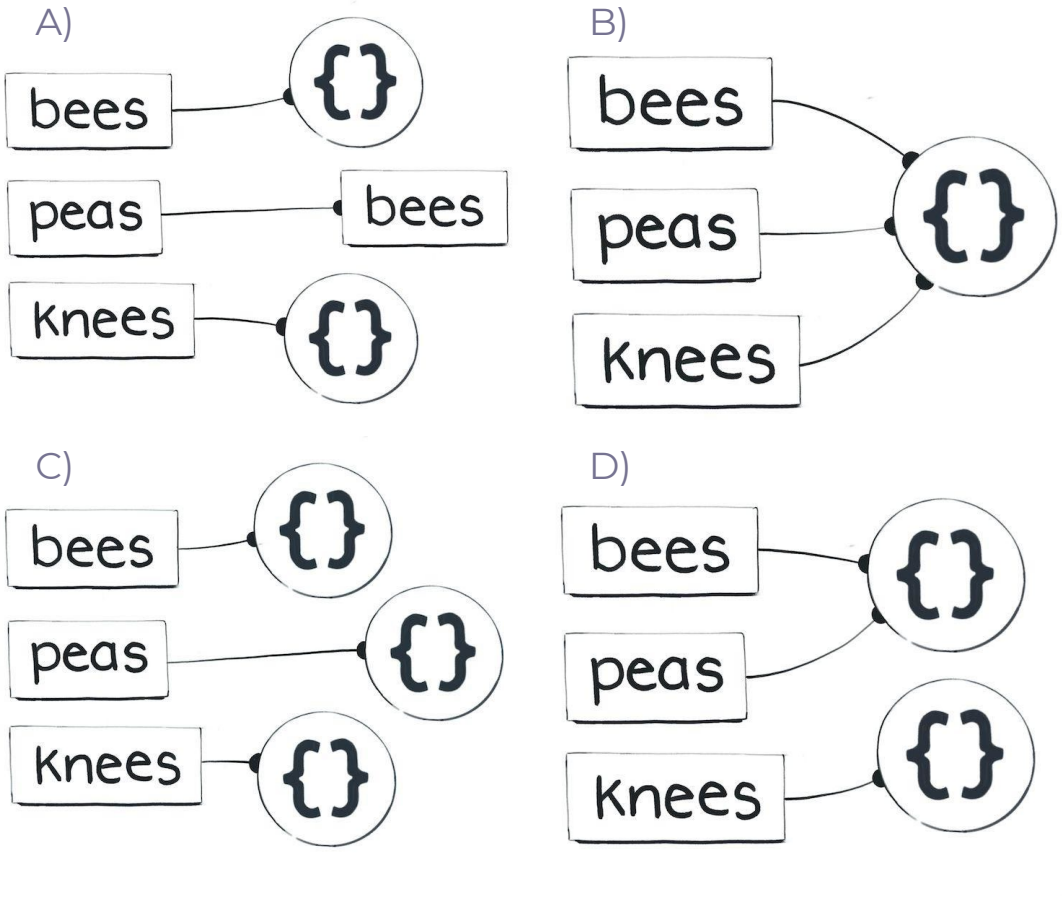


```
let bees = {};
let peas = bees;
let knees = {};
```

Exercises

Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let bees = {};
let peas = bees;
let knees = {};
```



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

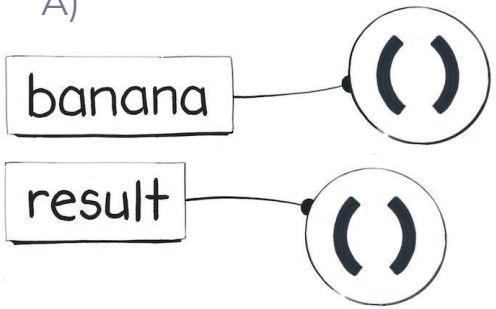
```
let banana = function() { return 2 + 2; };
let result = banana;
```

Exercises

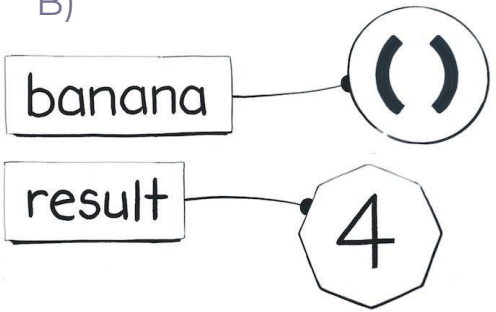
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let banana = function() { return 2 + 2; };
let result = banana;
```

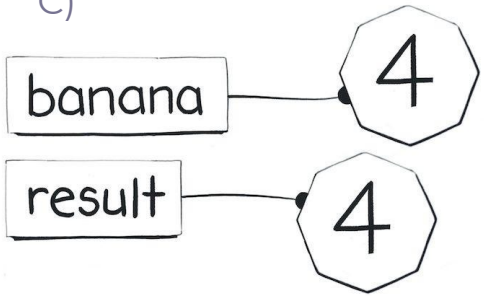
A)



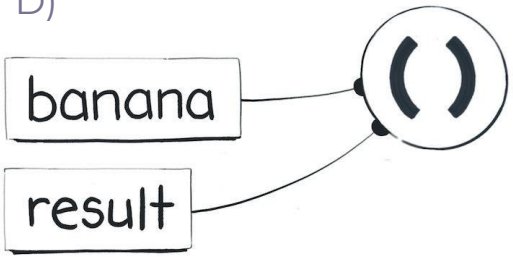
B)



C)



D)



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

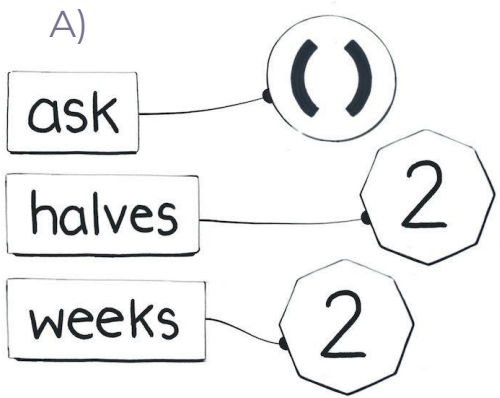
```
let ask = function() { return 2; };
let halves = ask();
let weeks = ask();
```

Exercises

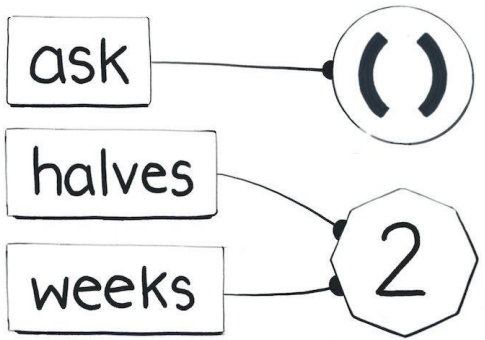
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let ask = function() { return 2; };
let halves = ask();
let weeks = ask();
```

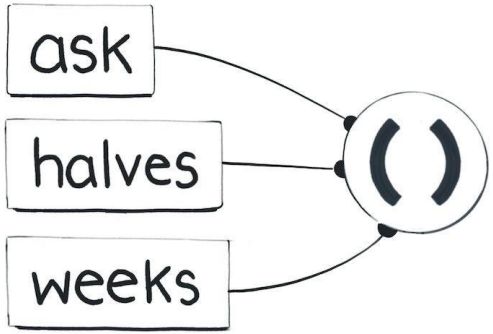
A)



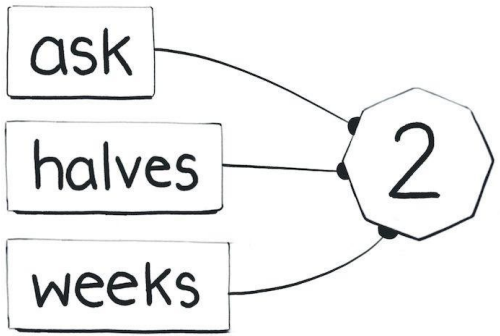
B)



C)

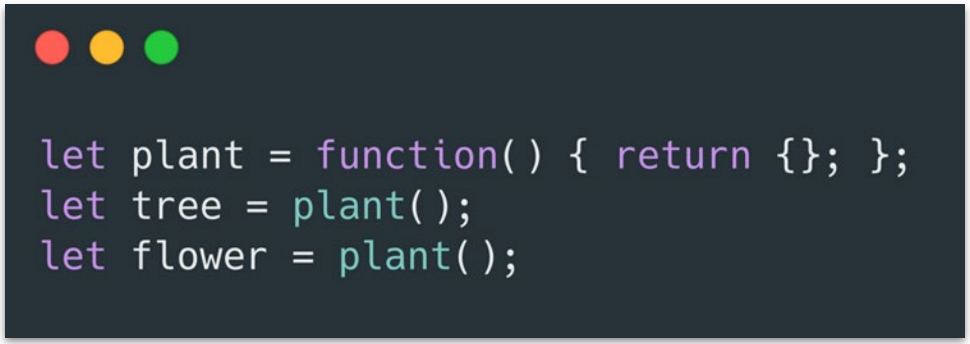


D)



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

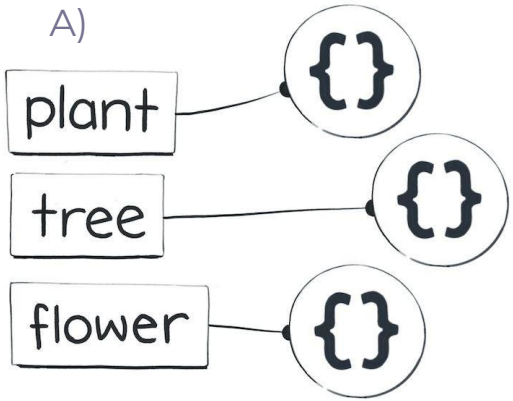


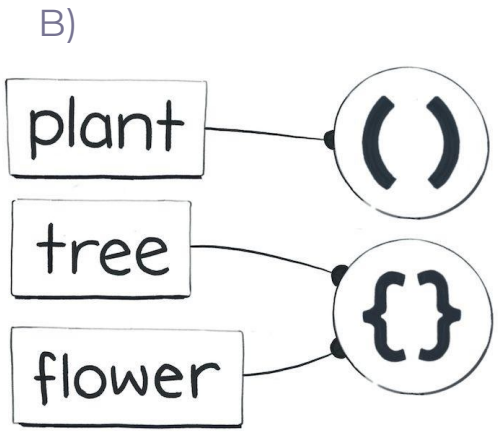
```
let plant = function() { return {}; };
let tree = plant();
let flower = plant();
```

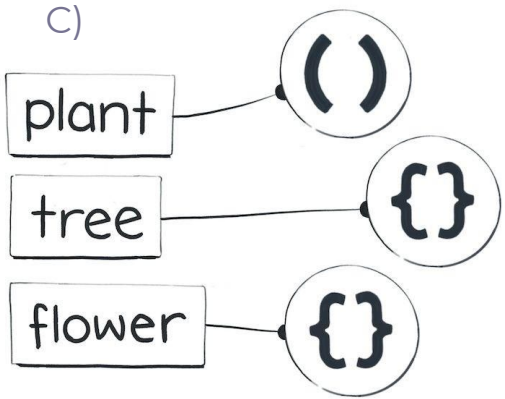
Exercises

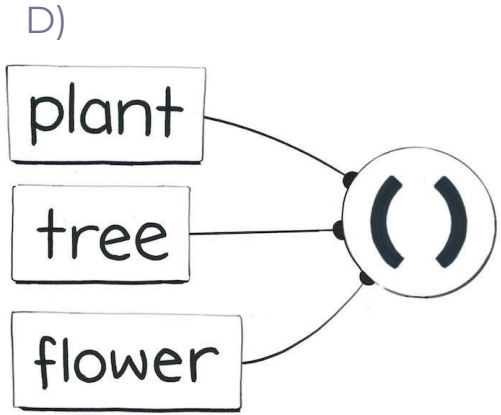
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let plant = function() { return {}; };
let tree = plant();
let flower = plant();
```









Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

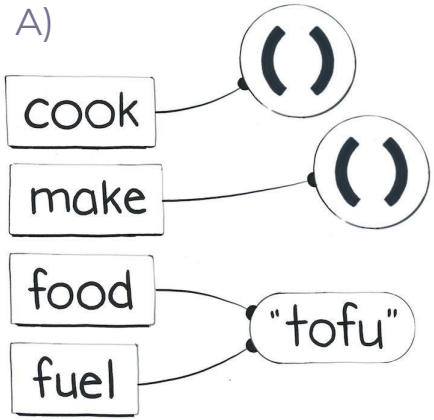
```
let cook = function() { return 'tofu'; };
let make = function() { return 'tofu'; };
let food = cook();
let fuel = make();
```

Exercises

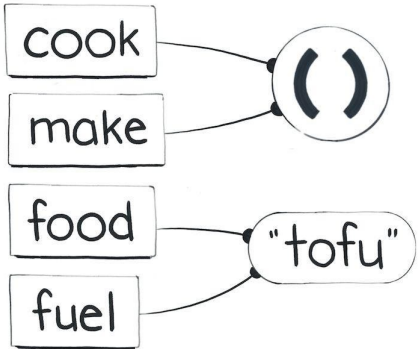
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let cook = function() { return 'tofu'; };
let make = function() { return 'tofu'; };
let food = cook();
let fuel = make();
```

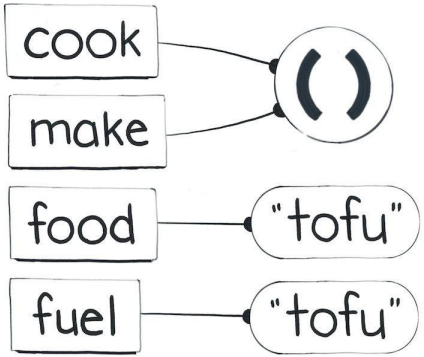
A)



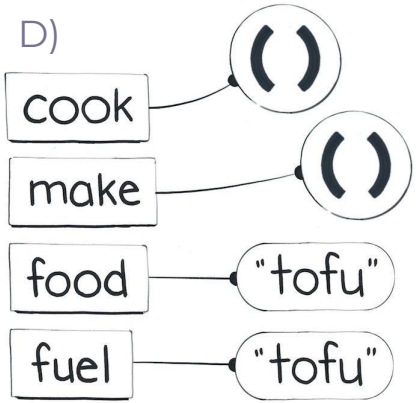
B)



C)



D)



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

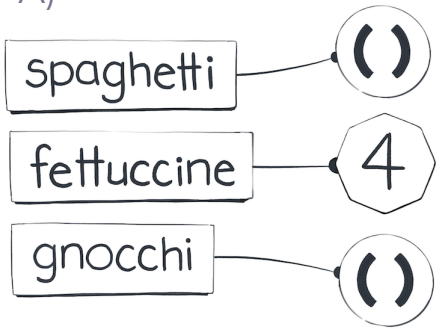
```
let spaghetti = function() { return 2 + 2 };
let fettuccine = spaghetti;
let gnocchi = function() { return 2 + 2 };
```

Exercises

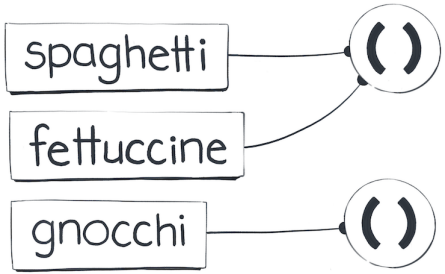
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let spaghetti = function() { return 2 + 2 };
let fettuccine = spaghetti;
let gnocchi = function() { return 2 + 2 };
```

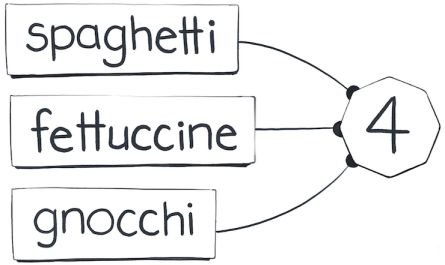
A)



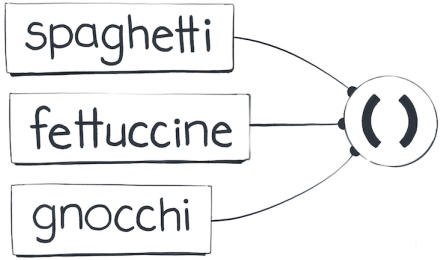
B)

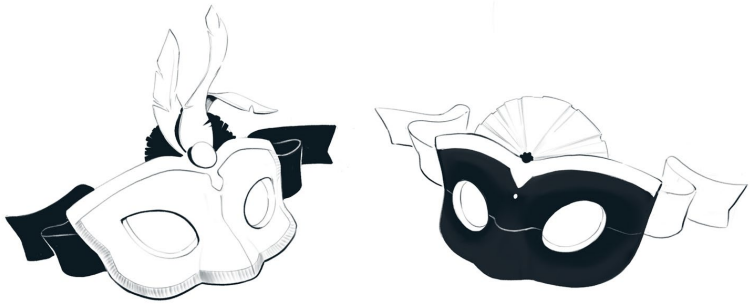


C)



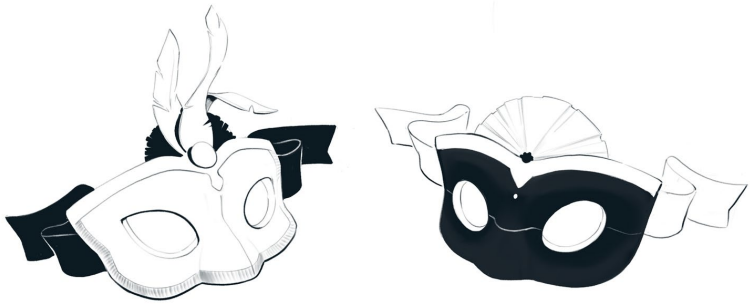
D)





Kinds of Equality

- **Strict Equality:**
  `a === b` (triple equals).
- **Loose Equality:**
  `a == b` (double equals).
- **Same Value Equality:**
  `Object.is(a, b)`.



If you don’t have a clear mental model of equality in JavaScript, every day is like a carnival – and not in a good way.

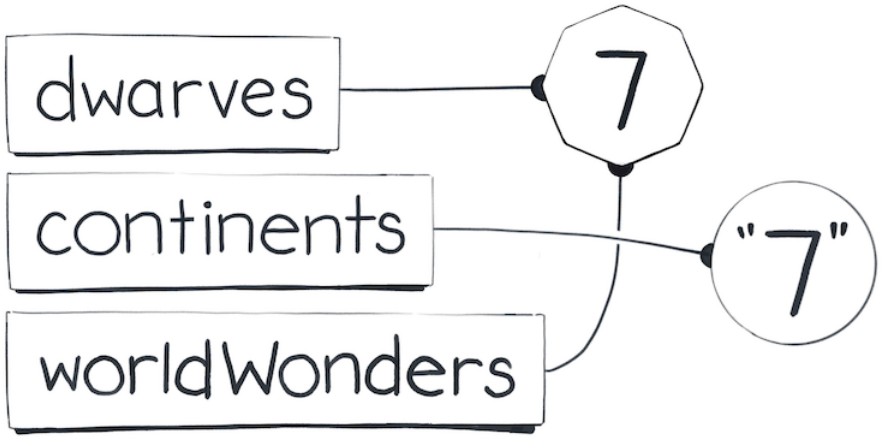
Same Value Equality: Object.is(a, b)

In JavaScript, Object.is(a, b) tells us if a and b are the same value:

```
console.log(Object.is(2, 2)); // true

console.log(Object.is({}, {})); // false
```

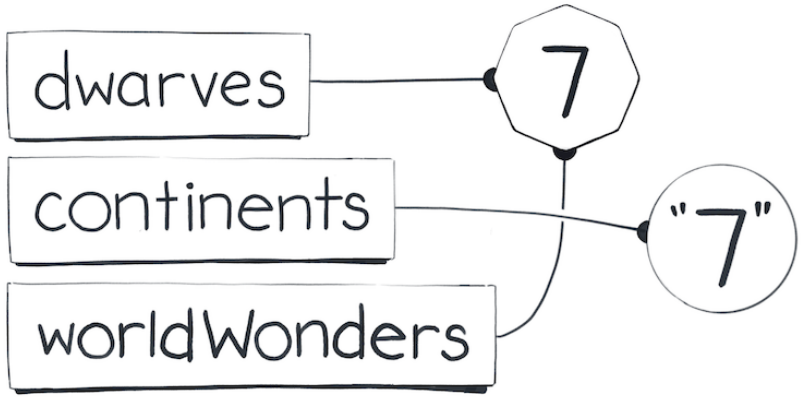
Check Your Intuition

```
let dwarves = 7;

let continents = '7';

let worldWonders = 3 + 4;
```



Try to answer these questions using the diagram:

```
console.log(Object.is(dwarves, continents)); // ?

console.log(Object.is(continents, worldWonders)); // ?

console.log(Object.is(worldWonders, dwarves)); // ?
```



This was not a trick question!

1. Object.is(dwarves, continents) is **false** because dwarves and continents **point at different values**.
2. Object.is(continents, worldWonders) is **false** because continents and worldWonders **point at different values**.
3. Object.is(worldWonders, dwarves) is **true** because worldWonders and dwarves **point at the same value**.

What About Objects?

```
let banana = {};

let cherry = banana;

let chocolate = cherry;

cherry = {};
```

```
console.log(Object.is(banana, cherry)); // ?

console.log(Object.is(cherry, chocolate)); // ?

console.log(Object.is(chocolate, banana)); // ?
```

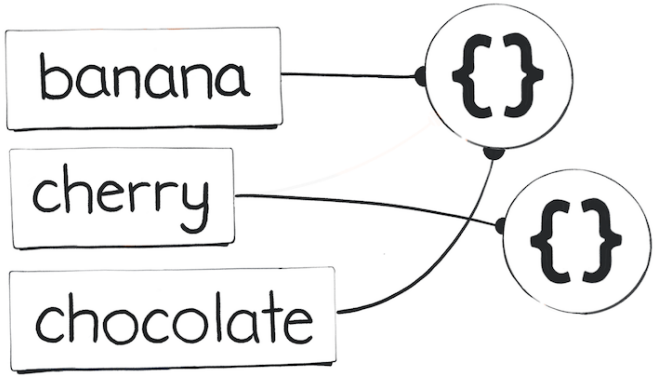
What About Objects?

```
let banana = {};
let cherry = banana;
let chocolate = cherry;
cherry = {};
```

What About Objects?

```
let banana = {};
let cherry = banana;
let chocolate = cherry;
cherry = {};
```

1. **let banana = {};**
    - Declare a banana variable.
    - Create a new object value {}.
    - Point banana variable's wire to it.
2. **let cherry = banana;**
    - Declare a cherry variable.
    - Point cherry's wire to where banana is pointing.
3. **let chocolate = cherry;**
    - Then, we declare a chocolate variable.
    - Point chocolate's wire to where cherry is pointing.
4. **cherry = {};**
    - Create a new object value {}.
    - Point cherry's wire to it.



1. Object.is(banana, cherry) is **false** because banana and cherry **point at different values**.
2. Object.is(cherry, chocolate) is **false** because cherry and chocolate **point at different values**.
3. Object.is(chocolate, banana) is **true** because chocolate and banana **point at the same value**.

Strict Equality: a === b

```
console.log(2 === 2); // true

console.log({} === {}); // false
```

There is also a corresponding opposite !== operator.

Same Value Equality vs Strict Equality

```
console.log( 2 === 2 )
```

Same Value Equality vs Strict Equality

```
console.log({}==={})
```

Exceptions to the rule

1. **NaN === NaN is false**, although they are the same value.
2. **-0 === 0 and 0 === -0 are true**, although they are different values.

First Special Case: NaN

As we've seen in *Counting the Values*, NaN is a special number that shows up when we do invalid math like 0 / 0:

```
let width = 0 / 0; // NaN
```

Further calculations with NaN will give you NaN again:

```
let height = width * 2; // NaN
```

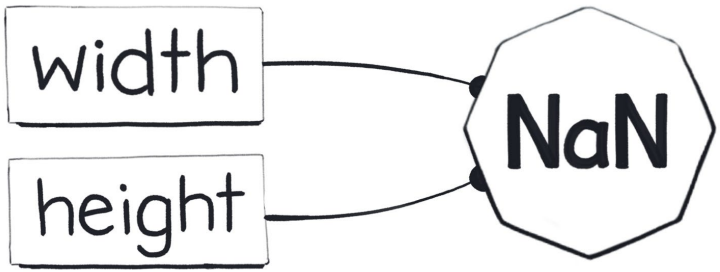
First Special Case: NaN

Remember that NaN === NaN is always false:

```
console.log(width === height); // false
```

However, NaN is the same value as NaN:

```
console.log(Object.is(width, height)); // true
```



First Special Case: NaN

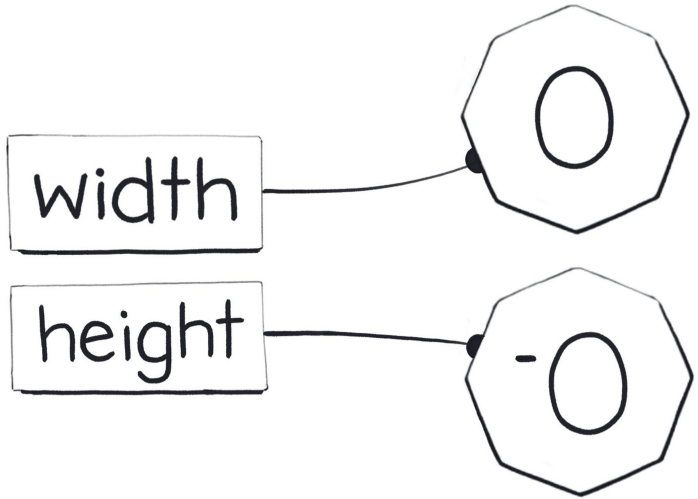
```
function resizeImage(size) {
  if (size === NaN) {
    // Doesn't work: the check is always false!
    console.log('Something is wrong.');
  }
  // ...
}
```

To check if size is NaN:

- Number.isNaN(size)
- Object.is(size, NaN)
- size !== size

Second Special Case: -0

In regular math, there is no such concept as “minus zero”, but it exists in floating point math for practical reasons.



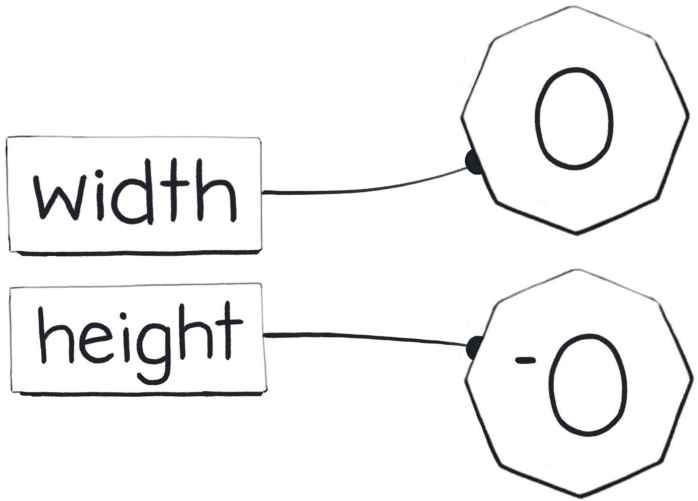
Second Special Case: -0

Both 0 === -0 and -0 === 0 are always true:

```
let width = 0; // 0

let height = -width; // -0

console.log(width === height); // true
```

However, 0 is a different value from -0:

```
console.log(Object.is(width, height)); // false
```



Loose Equality

Loose Equality (double equals) is the bogeyman of JavaScript.

Here's just a couple of examples to make your skin crawl:

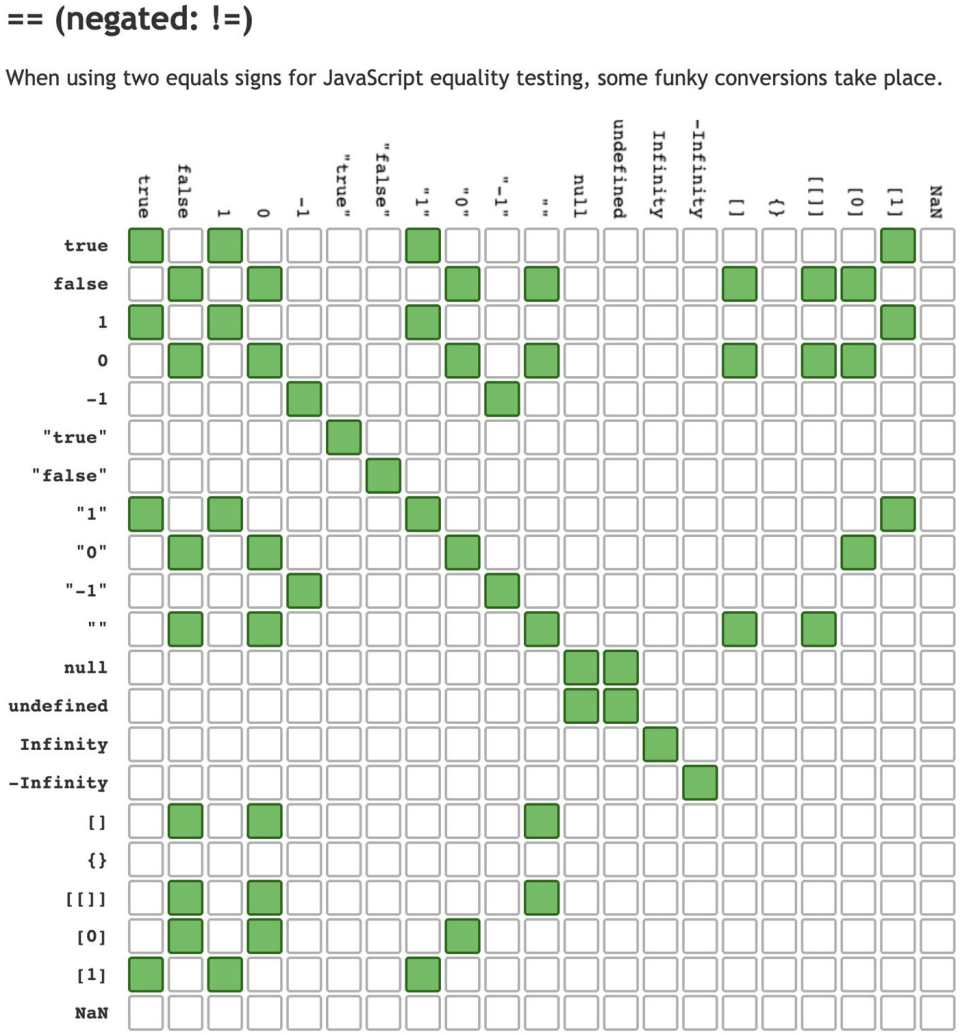
```
console.log([[]] == ''); // true

console.log(true == [1]); // true

console.log(false == [0]); // true
```

Loose Equality

https://dorey.github.io/JavaScript-Equality-Table/?ck_subscriber_id=848179219

Moral of the story:

Always use 3 equals unless you have a good reason to use 2.



Although Just JavaScript doesn't take strong opinions on what features you should or shouldn't use, we're not going to cover **Loose Equality** for now.

A Common Case

There is one usage of it that is relatively common and is worth knowing:

```
if (x == null) {
  // ...
}
```

This code is equivalent to writing:

```
if (x === null || x === undefined) {
  // ...
}
```

Thing to Remember

JavaScript has several kinds of equality.

- Same Value Equality
- Strict Equality
- Loose Equality.

Understanding Same Value Equality helps prevent bugs!

The same value cannot appear twice on a diagram of values and variables.

NaN === NaN is false, even though they are the same value.

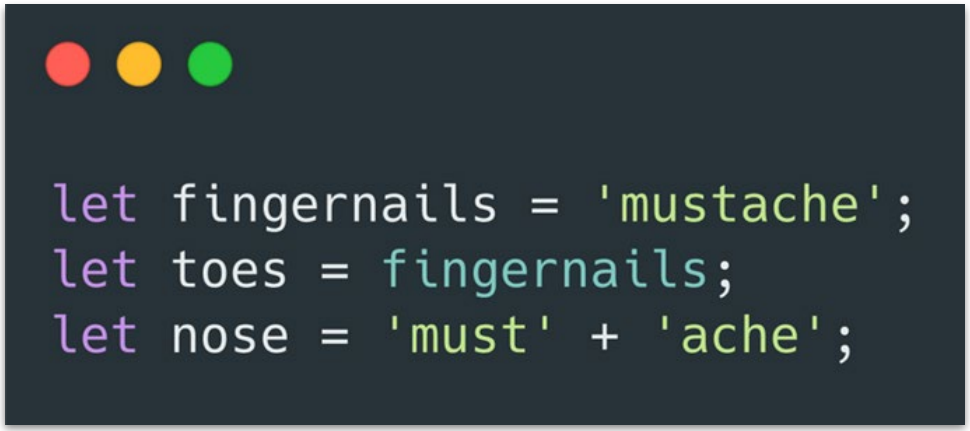
0 === -0 and -0 === 0 is true, but they are different values.

You can check whether x is NaN using Number.isNaN(x).

Loose Equality (==) is a set of arcane rules and is often avoided.

Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.



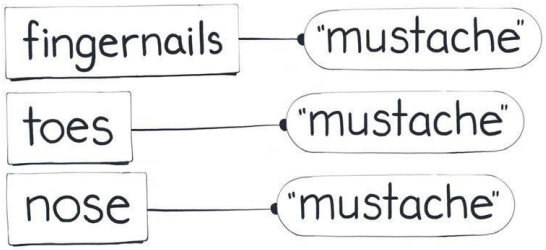
```
let fingernails = 'mustache';
let toes = fingernails;
let nose = 'must' + 'ache';
```

Exercises

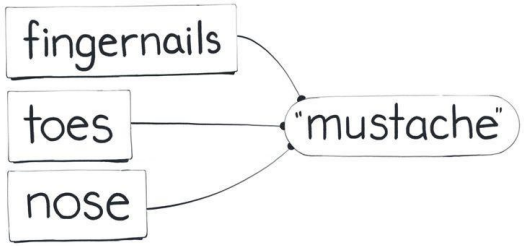
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let fingernails = 'mustache';
let toes = fingernails;
let nose = 'must' + 'ache';
```

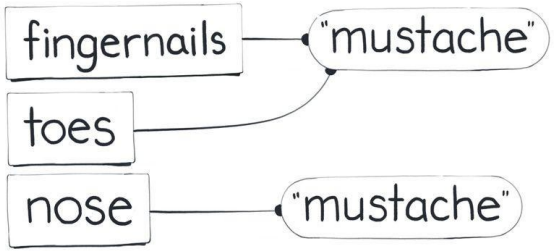
A)



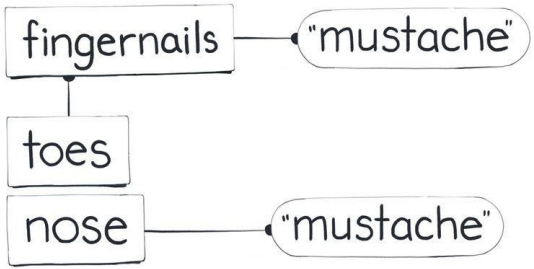
B)



C)



D)



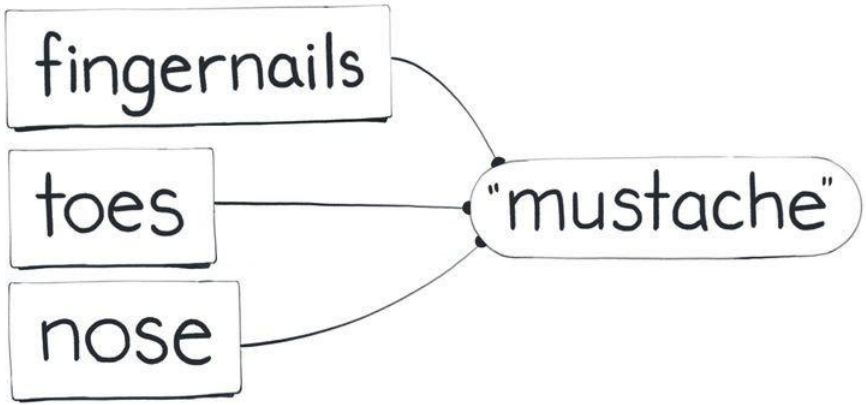
Now use this sketch to determine the output of:

1. `console.log(fingernails === toes);`

2. `console.log(toes === nose);`

3. `console.log(nose === fingernails);`

Hint: usually, a === b when a and b point to the same value.

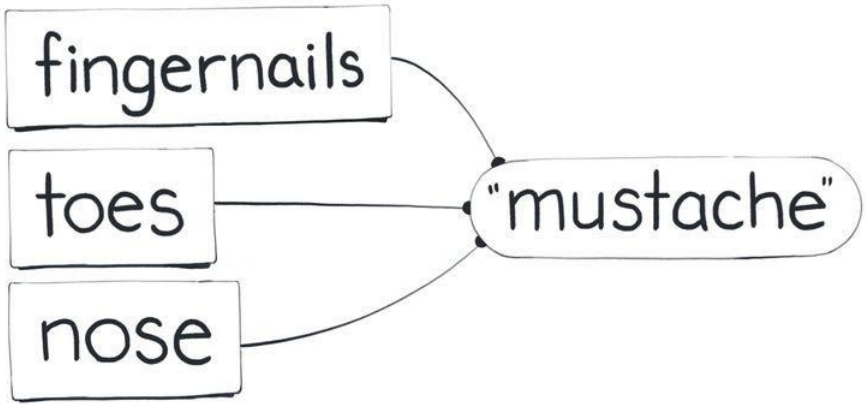


Answer: all of these examples print true.

1. `fingernails === toes`: This is `true` because fingernails and toes point at the same string.

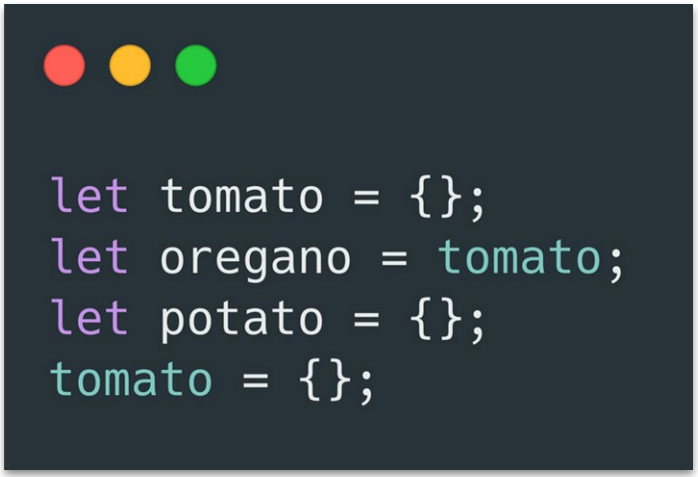
2. `toes === nose`: This is `true` because toes and nose point at the same string.

3. `nose === fingernails`: This is `true` because nose and fingernails point at the same string.



Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.



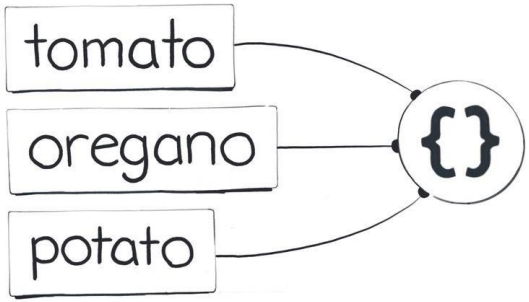
```
let tomato = {};
let oregano = tomato;
let potato = {};
tomato = {};
```

Exercises

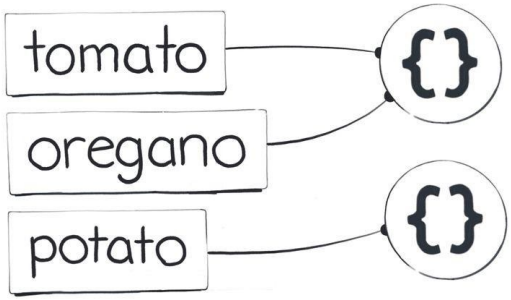
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let tomato = {};
let oregano = tomato;
let potato = {};
tomato = {};
```

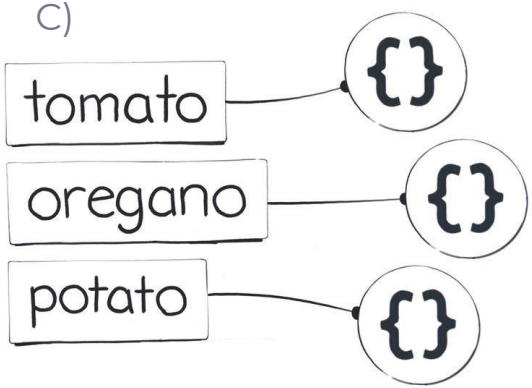
A)



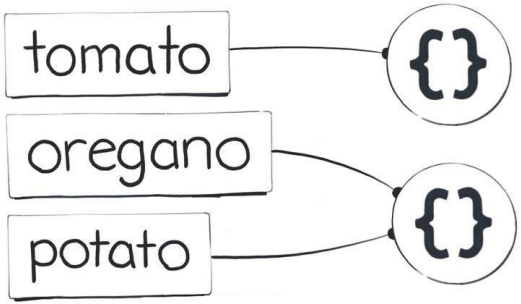
B)



C)

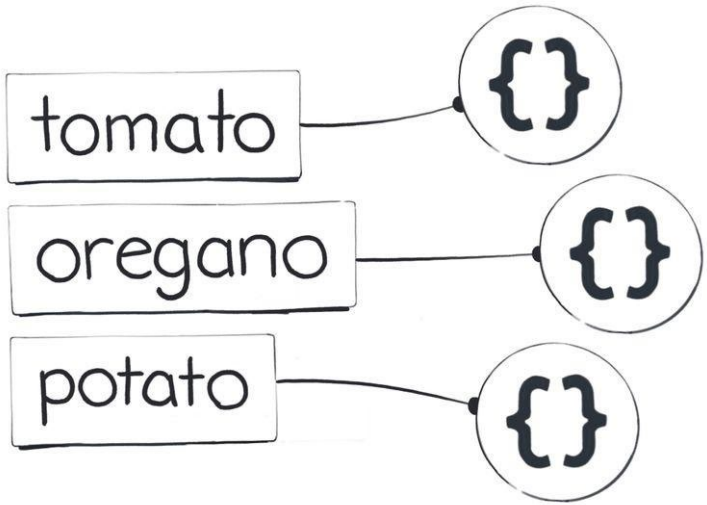


D)



Now use this sketch to determine the output of:

1. `console.log(tomato === oregano);;`
2. `console.log(oregano === potato);`
3. `console.log(potato === tomato);`

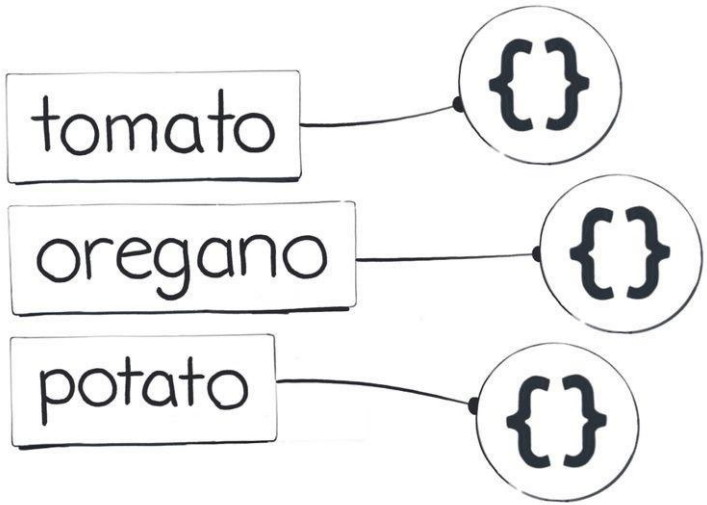


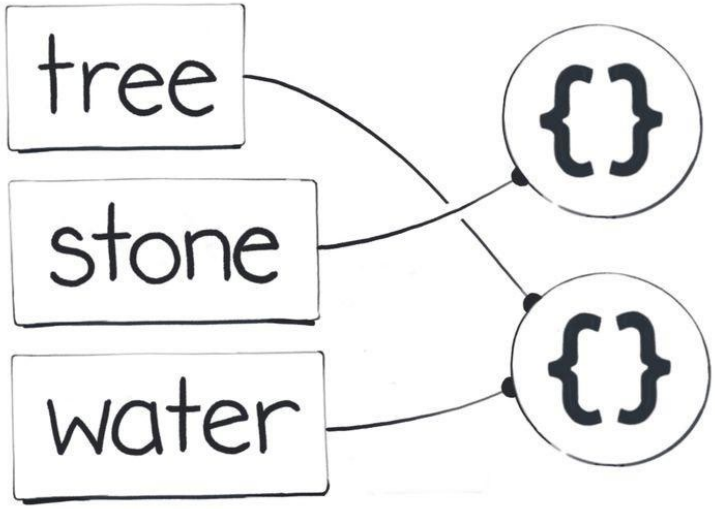
Answer: all of these examples print false.

1. `tomato === oregano`: This is `false` because *tomato* and *oregano* point at different objects.

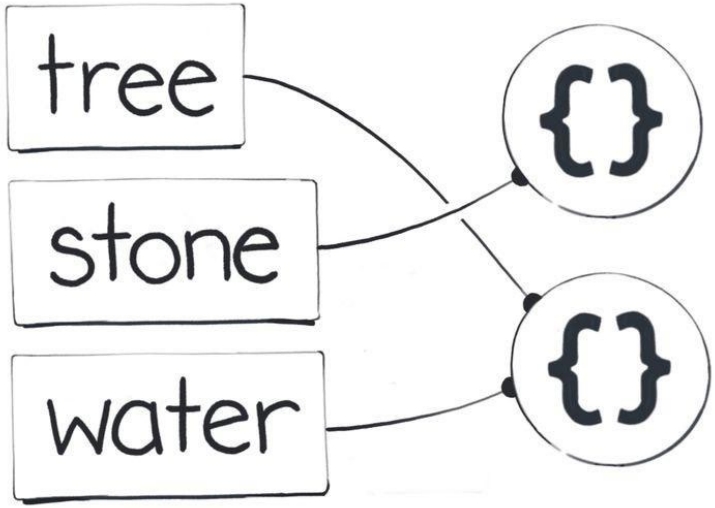
2. `oregano === potato`: This is `false` because *oregano* and *potato* point at different objects.

3. `potato === tomato`: This is `false` because *potato* and *tomato* point at different objects.





Look at this diagram of variables and values. Write three lines of code that, when executed, would lead to this diagram.




```
let tree = {};
let stone = {};
let water = tree;
```

```
let ticket = { id: 0 };
if (ticket === { id: 0 }) {
  console.log('Bad ticket');
}
```

The author of this code expects it to print a message. **Will the message appear?**

```
let ticket = { id: 0 };
if (ticket === { id: 0 }) {
  console.log('Bad ticket');
}
```

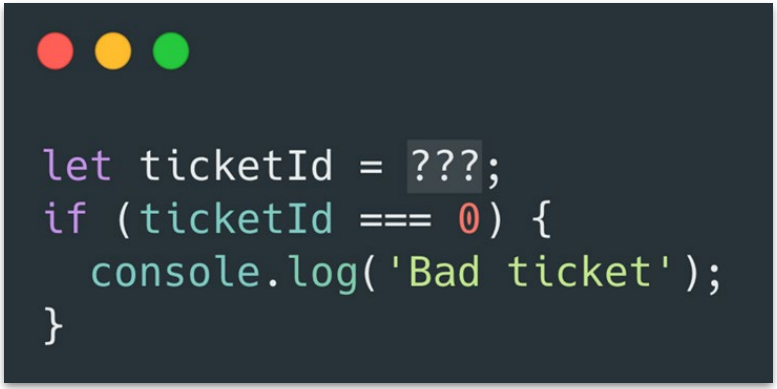
We have **different object values on each side of ===**. So the check will **not** pass, and the function will **not** print our message.

```
let ticketId = 0;
if (ticketId === 0) {
  console.log('Bad ticket');
}
```

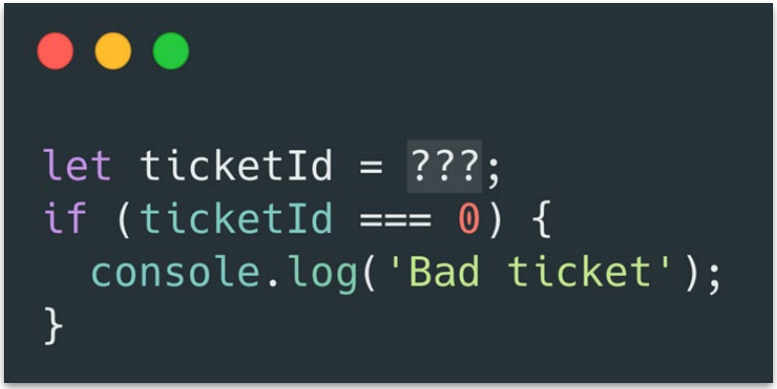
The author of this code expects it to print a message. **Will the message appear?**

```
let ticketId = 0;
if (ticketId === 0) {
  console.log('Bad ticket');
}
```

There is only one number value 0 in our universe, and **we have that same value on both sides of ===**. The check **will** pass, and the function **will** print our message.



Can you think of a different value than 0 that would also cause a message to be printed?

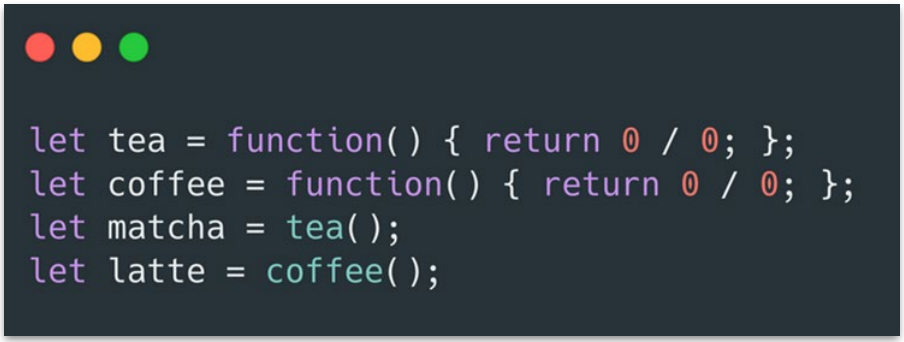


The other value that would cause the message to be printed is -0.

Even though 0 and -0 are two different values, === considers them equal.

Exercises

Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.



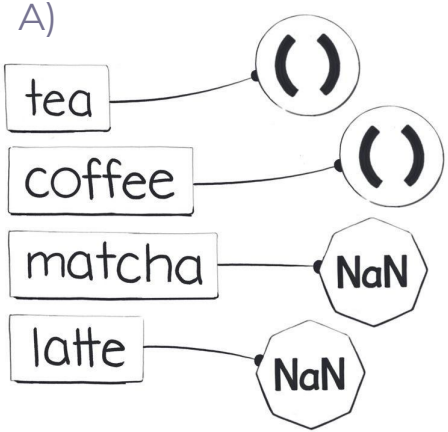
```
let tea = function() { return 0 / 0; };
let coffee = function() { return 0 / 0; };
let matcha = tea();
let latte = coffee();
```

Exercises

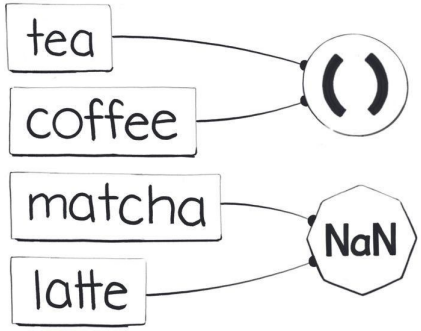
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let tea = function() { return 0 / 0; };
let coffee = function() { return 0 / 0; };
let matcha = tea();
let latte = coffee();
```

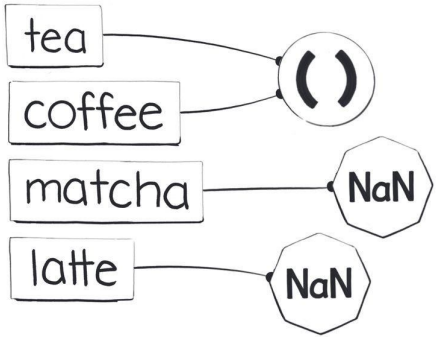
A)



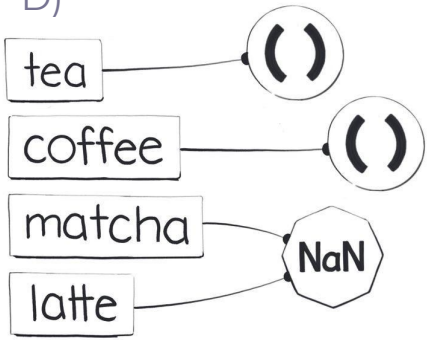
B)



C)

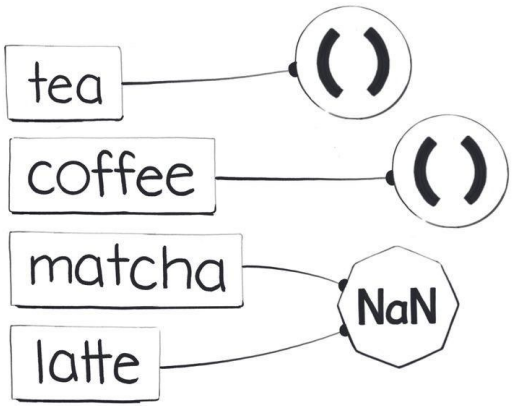


D)



Now use this sketch to determine the output of:

1. `console.log(tea === coffee);`
2. `console.log(Object.is(tea, coffee));`
3. `console.log(matcha === latte);`
4. `console.log(Object.is(matcha, latte));`



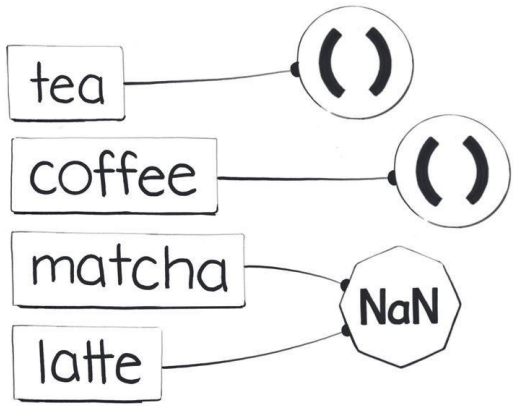
Answer: Only Object.is(matcha, latte) is true, all the other answers are false.

1. `tea === coffee` — This is false because tea and coffee point at two different function values.

2. `Object.is(tea, coffee)` — This is false because tea and coffee point at two different function values.

3. `matcha === latte` — This is false because both of them point at NaN, and NaN === NaN is always false. (If you forgot about this, you can re-read the section about special cases of Strict Equality.)

4. `Object.is(matcha, latte)` — This is true because matcha and latte point at the same NaN number value.



Coding Exercise

Write a function called **`strictEquals(a, b)`** that returns the same value as **`a === b`**. Your implementation must not use the **`===`** or **`!==`** operators.

Given the **`strictEquals`** function
When <A> and <B> are compared
Then the result will be <Result>

A	B	Result
`1`	`1`	`true`
`NaN`	`NaN`	`false // Rule Exception`
`0`	`-0`	`true // Rule Exception`
`-0`	`0`	`true // Rule Exception`
`1`	`'1'`	`false`
`true`	`false`	`false`
`false`	`false`	`true`
`'Water'`	`'oil'`	`false`