Properties

Meet Sherlock Holmes, a world-renowned detective from London:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```

Meet Sherlock Holmes, a world-renowned detective from London:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```

His friend John Watson has recently moved in to live with Sherlock:

```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};

let john = {
  surname: 'Watson',
  address: sherlock.address
};

john.surname = 'Lennon';

john.address.city = 'Malibu';
```

Write down your answers to these questions:

```
console.log(sherlock.surname); // ?
console.log(sherlock.address.city); //
?
console.log(john.surname); // ?
console.log(john.address.city); // ?
```

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};

let john = {
  surname: 'Watson',
  address: sherlock.address
};

john.surname = 'Lennon';

john.address.city = 'Malibu';
```

```
// "Holmes"
console.log(sherlock.surname);

// "Malibu"
console.log(sherlock.address.city);

// "Lennon"
console.log(john.surname);

// "Malibu"
console.log(john.address.city);
```

Properties

We create a new object value by writing {}:

```
let sherlock = {};
```

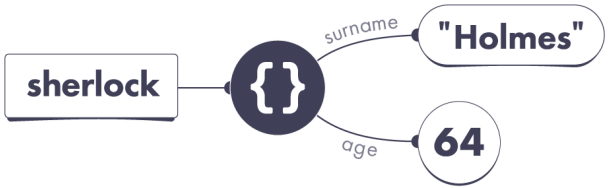
In our universe, it might look like this:

Objects are primarily useful to group related data together.

```
let sherlock = {

  surname: 'Holmes',

  age: 64,

};
```

In our JavaScript universe, both variables and properties act like “wires”.

The wires of properties start from objects rather than from our code:



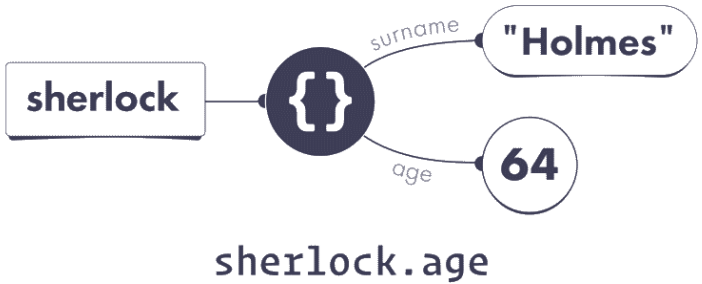
Properties don’t contain values —
Properties point at values!

Reading a Property

We can read a property's current value by using the "dot notation":

```
console.log(sherlock.age); // 64
```

Here, `sherlock.age` an expression, a question to the JavaScript universe. To answer it, JavaScript first follows the `sherlock` wire:



Property Names

A single object can't have two properties with the same name.

The property names are always case-sensitive!

Property Names

If we don’t know a property name ahead of time but we have it in code as a string value, we can use the [] “bracket notation” to read it from an object:

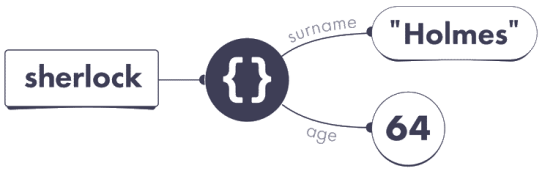
```
let sherlock = { surname: 'Holmes', age: 64 };

let propertyName = prompt('What do you want to know?');

alert(sherlock[propertyName]); // Read property by its name
```

Assigning to a Property

What happens when we assign a value to a property?

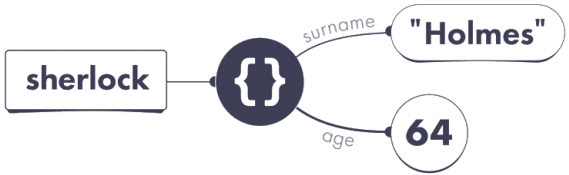
```
sherlock.age = 65;
```

First, we figure out which wire is on the left side: `sherlock.age`.



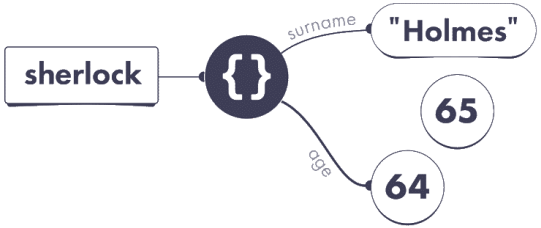

```
sherlock.age = 65
```

Next, we figure out which value is on the right side: 65.



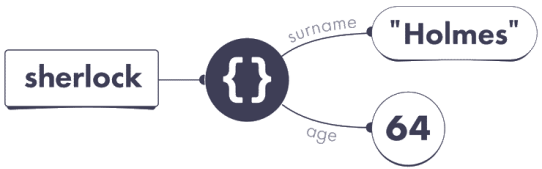

```
sherlock.age = 65
```

At last, we point the wire on the left side to the value on the right side:



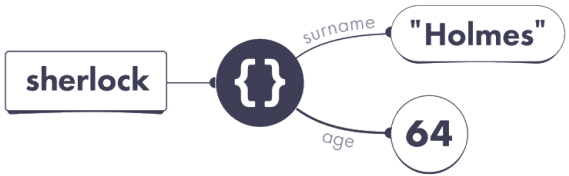

```
sherlock.age = 65
```

First, we figure out which wire is on the left side: `sherlock.age`.



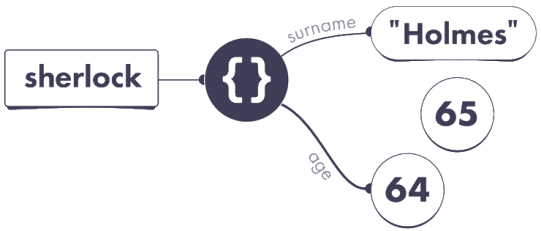

```
sherlock.age = 65
```

Next, we figure out which value is on the right side: 65.




```
sherlock.age = 65
```

At last, we point the wire on the left side to the value on the right side:




```
sherlock.age = 65
```

Missing Properties

What happens if we read a property that doesn't exist:

```
let sherlock = { surname: 'Holmes', age: 64 };

console.log(sherlock.boat); // ?
```

Missing Properties

JavaScript follows certain rules to decide which value to “answer” us with.

1. Figure out the value of the part before the dot (.).
2. If that value is null or undefined, throw an error immediately.
3. Check whether a property with that name exists in our object.
   a. If **it exists**, answer with the value this property points to.
   b. If **it doesn’t exist**, answer with the undefined value.

Missing Properties

What happens if we read a property that doesn't exist:

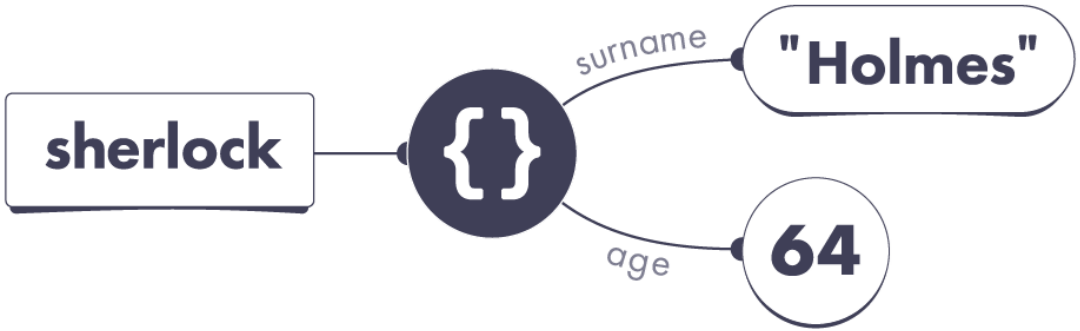
```
let sherlock = { surname: 'Holmes', age: 64 };

console.log(sherlock.boat); // ?
```

Missing Properties

What happens if we read a property that doesn't exist:

```
let sherlock = { surname: 'Holmes', age: 64 };

console.log(sherlock.boat); // undefined
```

This does not mean that our object has a boat property pointing to **`undefined`**!



It only has two properties, and neither of them is called boat.

What happens if we run this code?

```
let sherlock = { surname: 'Holmes', age: 64 };

console.log(sherlock.boat.name); // ?
```

Answer: Calculating `sherlock.boat.name` throws an error:

- We need to first figure out the value of sherlock.boat.
    - To do that, we need to figure out the value of sherlock.
        - The wire from sherlock variable leads to an object.
        - Therefore, the value of sherlock is that object.

    - An object is not null or undefined, so we keep going.
    - That object **does not** have a boat property.
    - Therefore, the value of sherlock.boat is undefined.

- We've got undefined on the left side of the dot (.).
- The rules say that null or undefined on the left side is an error.

Answer: Calculating sherlock.boat.name throws an error:

```
let sherlock = { surname: 'Holmes', age: 64 };

console.log(sherlock.boat); // undefined

console.log(sherlock.boat.name); // TypeError!

console.log(sherlock.boat?.name); // undefined
```

Thing to Remember

Properties are wires — a bit like variables.

- They both point at values.
- Unlike variables, properties **start from objects** in our universe.

Properties have names.

Generally, an assignment has three steps:

- Figure out which wire is on the left.
- Figure out which value is on the right.
- Point that wire to that value.

An expression like `obj.property` is calculated in three steps:

- Figure out which value is on the left.
- If it's null or undefined, throw an error.
- If that property exists, the result is the value its wire points to.
- If that property doesn't exist, the result is undefined.

Exercises

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

Exercises

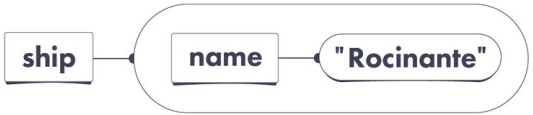
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let ship = { name: 'Rocinante' };
```

A)

B)

C)



D)

Exercises

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

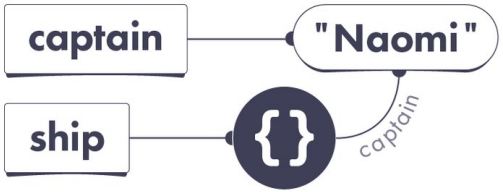
```
let captain = 'Jim';
let ship = { captain: captain };
captain = 'Naomi';
```

Exercises

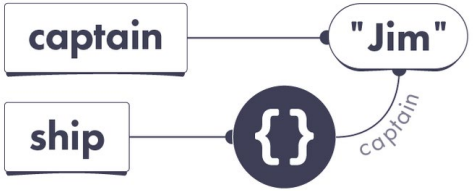
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let captain = 'Jim';
let ship = { captain: captain };
captain = 'Naomi';
```

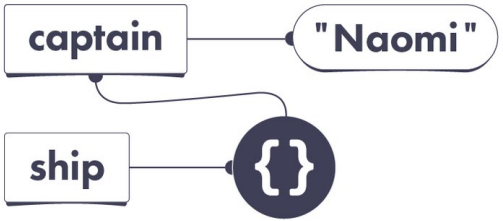
A)



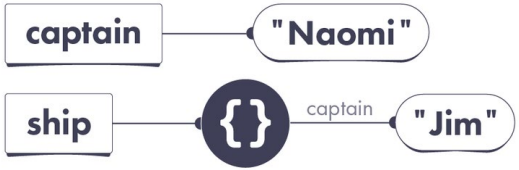
B)



C)



D)



Exercises

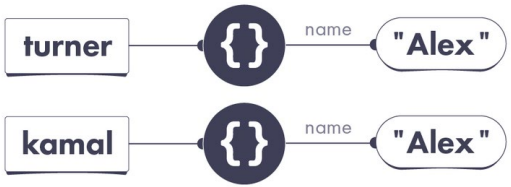
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

Exercises

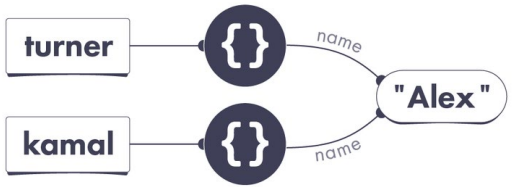
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let turner = { name: 'Alex' };
let kamal = { name: 'Alex' };
```

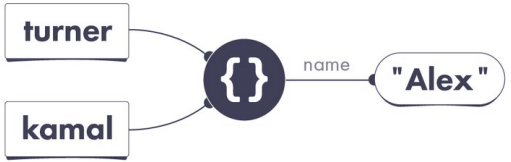
A)



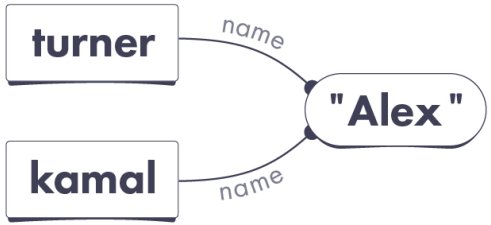
B)



C)



D)



Exercises

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

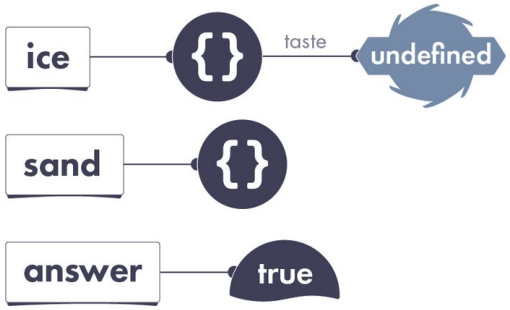
```
let ice = { taste: undefined };
let sand = {};
let answer = ice.taste === sand.taste;
```

Exercises

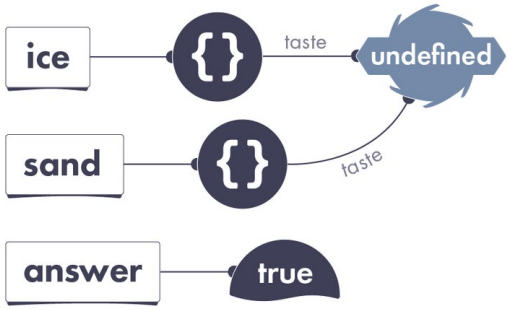
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let ice = { taste: undefined };
let sand = {};
let answer = ice.taste === sand.taste;
```

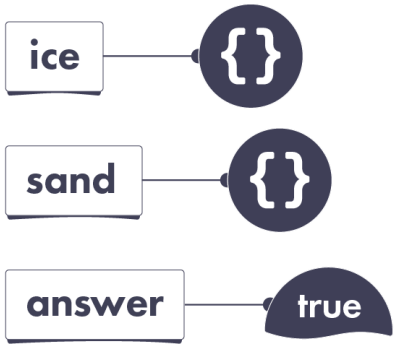
A)



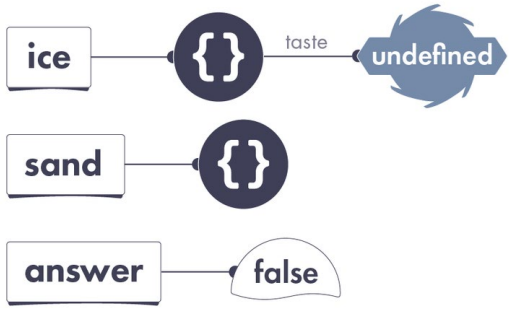
B)



C)



D)



Exercises

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

```
let ship = {
  pilot: { name: 'Jim' }
};
```

Exercises

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

```
let ship = {
  pilot: { name: 'Jim' }
};
```

A)

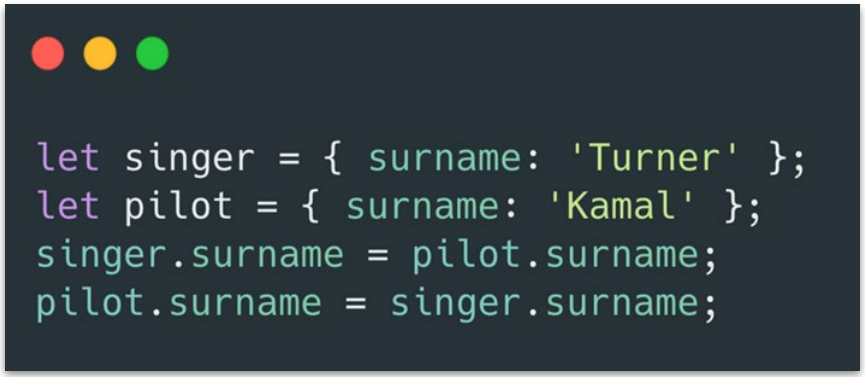
B)

C)

D)

Exercises

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



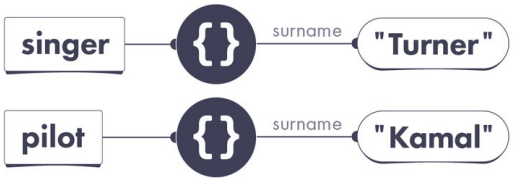
```
let singer = { surname: 'Turner' };
let pilot = { surname: 'Kamal' };
singer.surname = pilot.surname;
pilot.surname = singer.surname;
```

Exercises

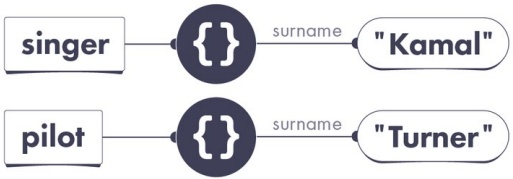
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let singer = { surname: 'Turner' };
let pilot = { surname: 'Kamal' };
singer.surname = pilot.surname;
pilot.surname = singer.surname;
```

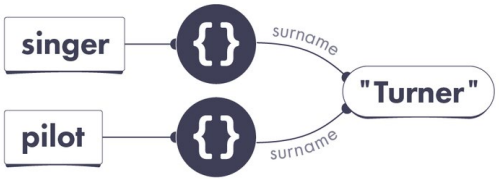
A)



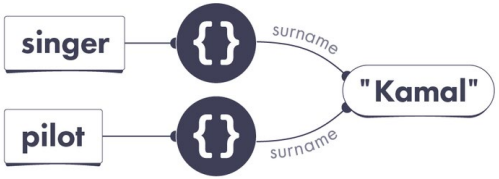
B)



C)

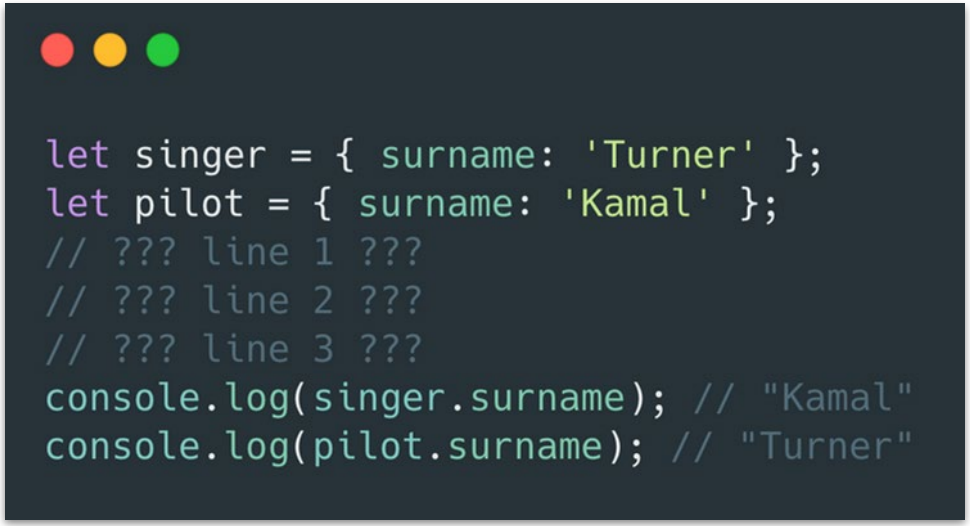


D)



Exercises

Can you figure out the three lines of code necessary to swap the values of the surname properties of these two objects?

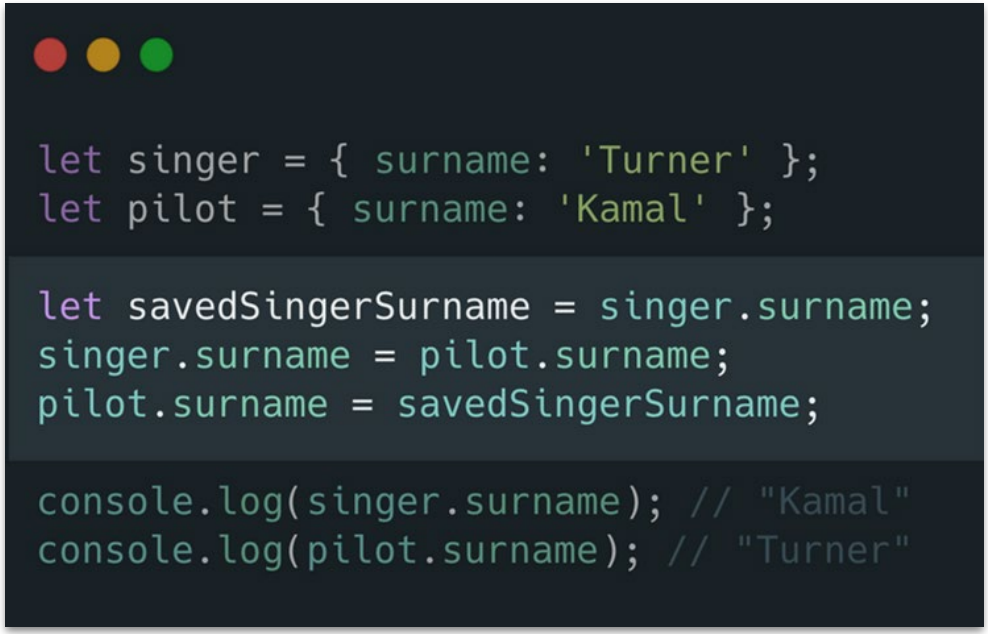


```
let singer = { surname: 'Turner' };
let pilot = { surname: 'Kamal' };
// ??? line 1 ???
// ??? line 2 ???
// ??? line 3 ???
console.log(singer.surname); // "Kamal"
console.log(pilot.surname); // "Turner"
```

Answer

We can't point two wires to different values in one line.

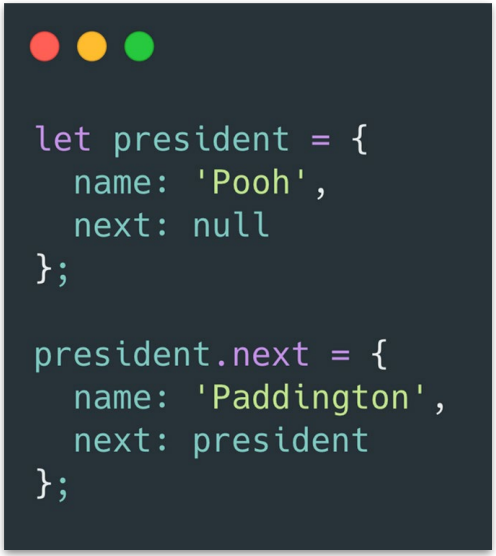
We can only change where one of them points at a time.



```
let singer = { surname: 'Turner' };
let pilot = { surname: 'Kamal' };

let savedSingerSurname = singer.surname;
singer.surname = pilot.surname;
pilot.surname = savedSingerSurname;

console.log(singer.surname); // "Kamal"
console.log(pilot.surname); // "Turner"
```

Exercises

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



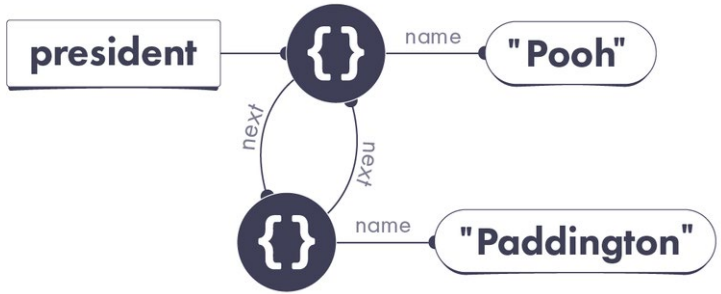
```
let president = {
  name: 'Pooh',
  next: null
};

president.next = {
  name: 'Paddington',
  next: president
};
```

Answer

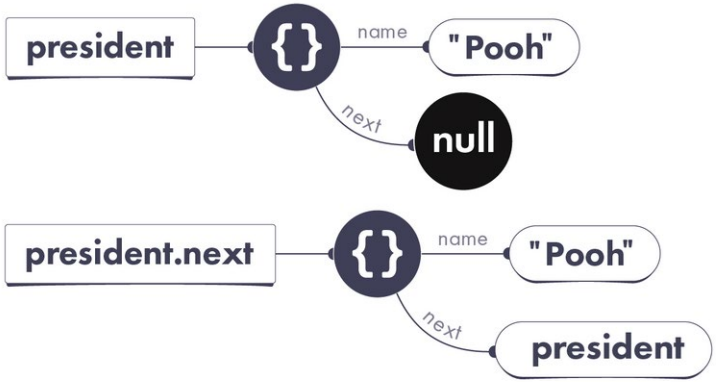
Which one of these diagrams best matches your sketch and our mental model after that code runs?

```
let president = {
  name: 'Pooh',
  next: null
};

president.next = {
  name: 'Paddington',
  next: president
};
```

A)



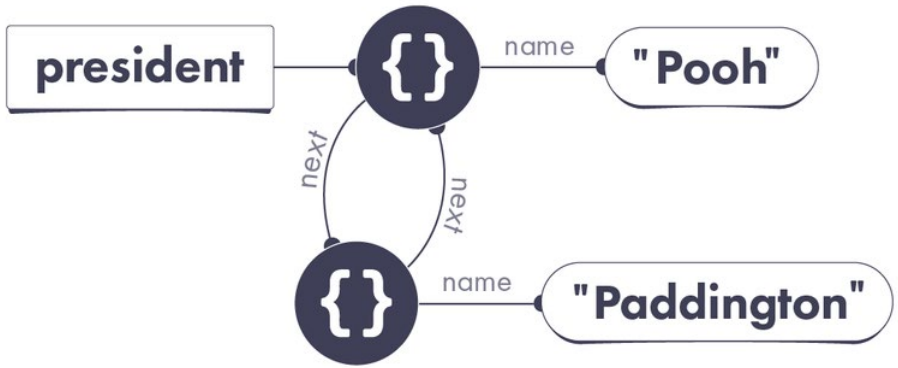
B)



Exercises

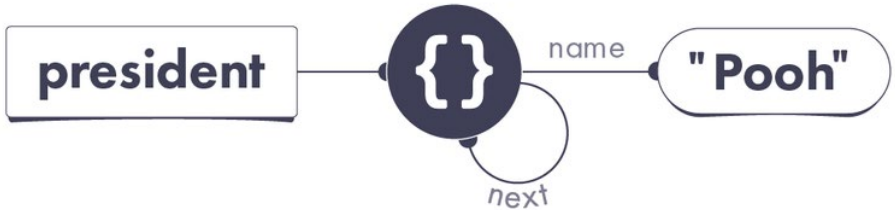
Now use this diagram to answer a question:

```
console.log(president.next.next.next.name);
```



Exercises

Write the code that produces this diagram.



Answer

Note that `let president = { next: president }` would **not** work.

Assignment happens in three steps:

(1) find the wire on the left,

(2) find the value on the right,

(3) point the wire to that value.

```
let president = { name: 'Pooh' };
president.next = president;
```

Exercises

What is the result of running this code?

```
let station = {
  Owner: { name: 'Fred' }
};
let name = station.owner.name;
console.log(name === station.Owner.name);
```

Answer

If station.owner is `undefined`, then it cannot appear on the left side of the dot.

Calculating `station.owner.name` produces an error and the code stops executing.

```
let station = {
  Owner: { name: 'Fred' }
};
let name = station.owner.name;
console.log(name === station.Owner.name);
```

Mutation

Step 1: Declaring the sherlock Variable

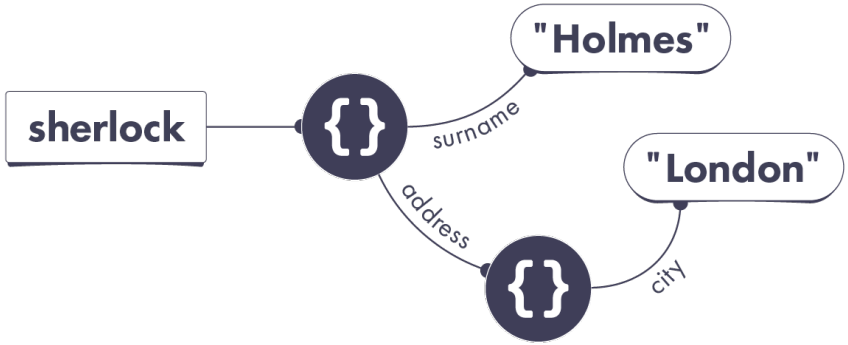
Sketch a diagram of variables and values after this snippet of code:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```

Step 1: Declaring the sherlock Variable

Sketch a diagram of variables and values after this snippet of code:

```
let sherlock = {
  surname: 'Holmes',
  address: { city: 'London' }
};
```



```
let sherlock = {
  surname: 'Holmes',
  address: { city:'London' }
};
```

Notice that we have two separate objects.

An object cannot be “inside” of other object!

Step 2: Declaring the `john` Variable

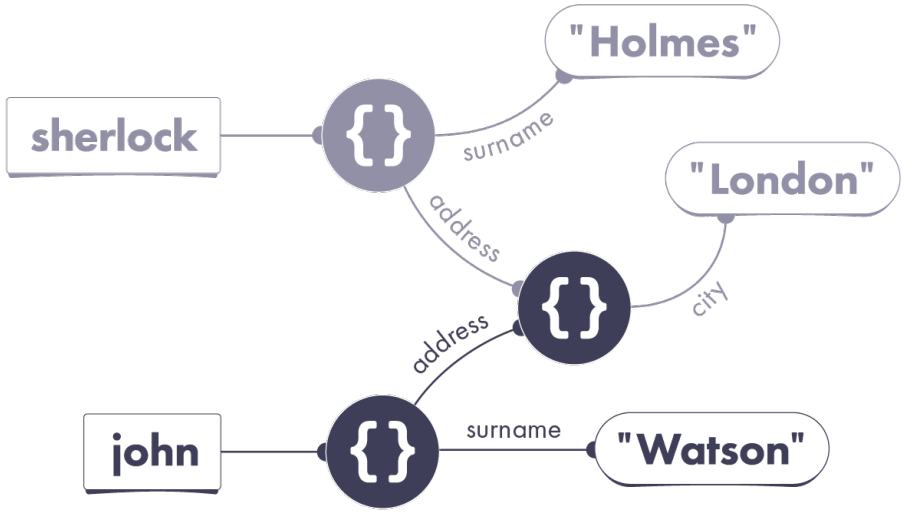
Sketch a diagram of variables and values after this snippet of code:

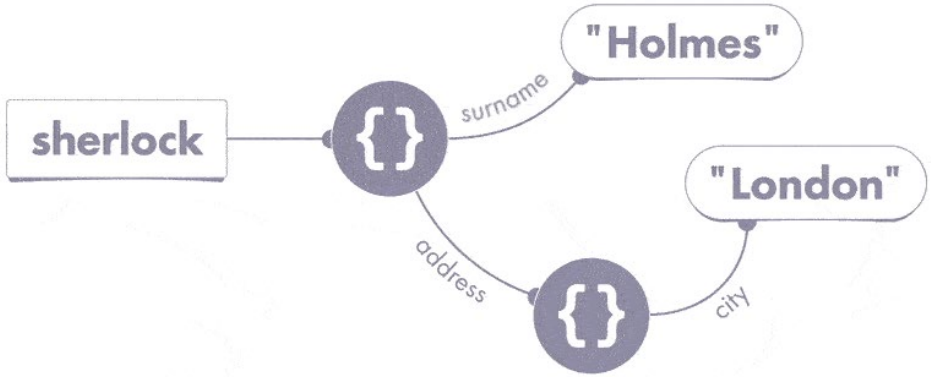
```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

Step 2: Declaring the `john` Variable

Sketch a diagram of variables and values after this snippet of code:

```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

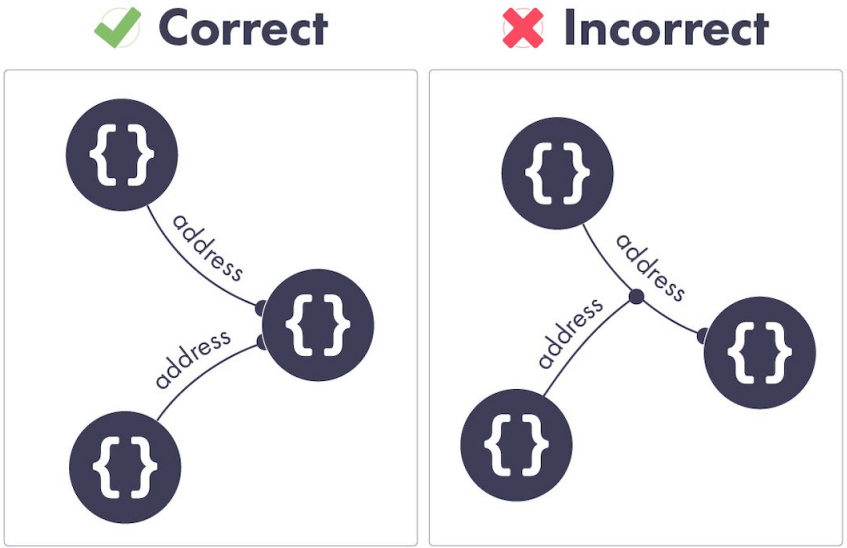





```
let john = {
  surname: 'Watson',
  address: sherlock.address
};
```

Notice that Properties Always Point at Values.

Properties Always Point at Values



Step 3: Changing the Properties

John has an identity crisis, and gets sick of the London drizzle. He decides to change his name and move to Malibu.

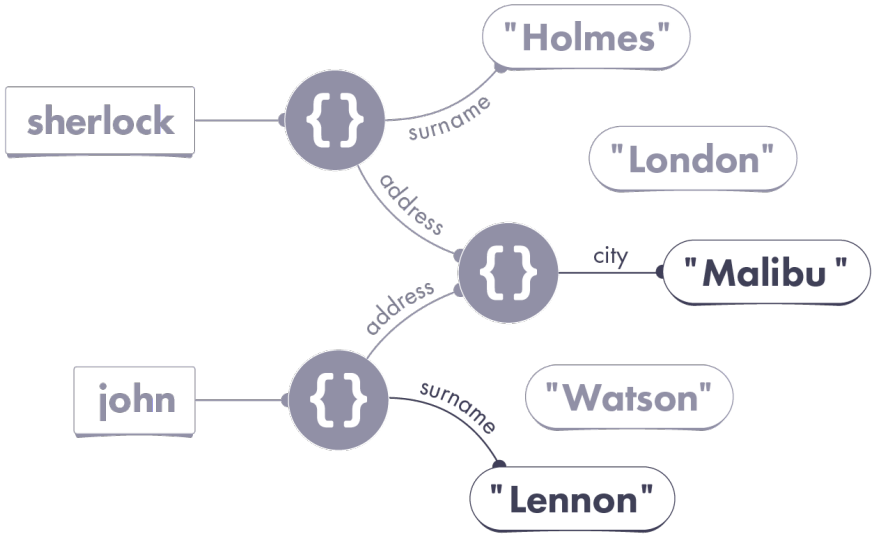
```
john.surname = 'Lennon';

john.address.city = 'Malibu';
```

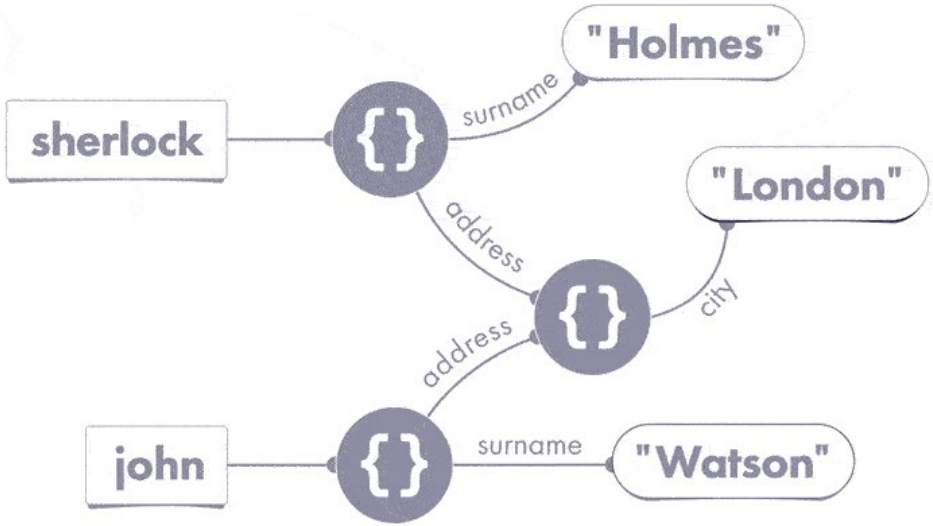
How do we change the diagram to reflect it?

Step 3: Changing the Properties

John has an identity crisis, and gets sick of the London drizzle. He decides to change his name and move to Malibu.

```
john.surname = 'Lennon';

john.address.city = 'Malibu';
```






```
john.surname = 'Lennon';
john.address.city = 'Malibu';
```

We figure out the wire, then the value, and finally point the wire to that value.

Mutation

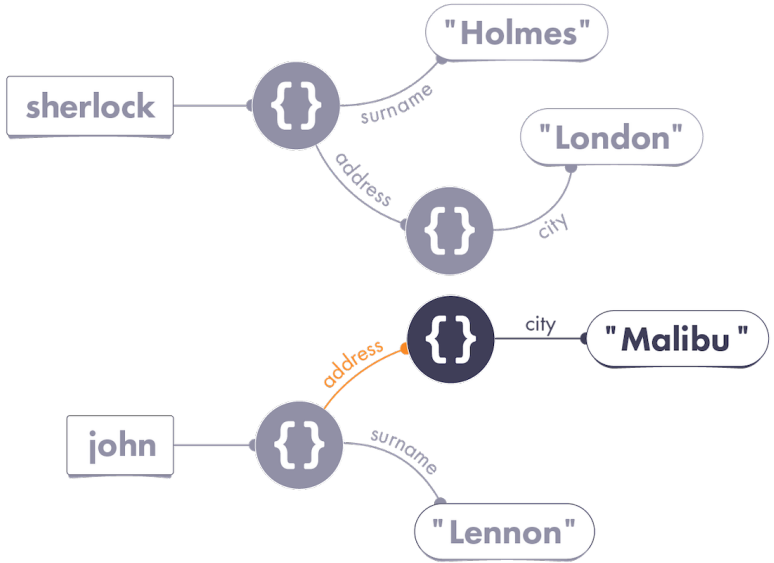
Mutation is a fancy way of saying “change”.

Possible Solution: Mutating Another Object

```
// Replace Step 3 with this code:

john.surname = 'Lennon';

john.address = { city: 'Malibu' };
```

Possible Solution: Mutating Another Object

```
// Replace Step 3 with this code:

john.surname = 'Lennon';

john.address = { city: 'Malibu' };
```

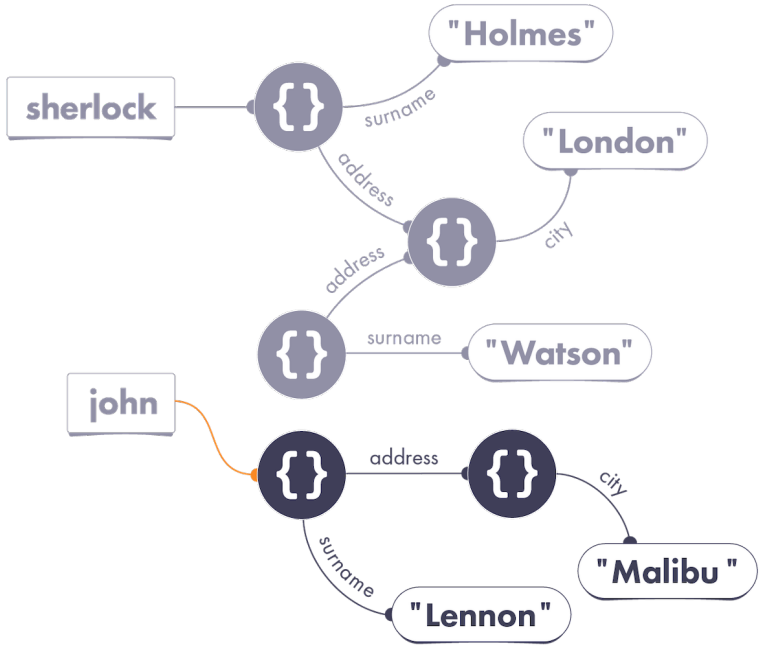


Alternative Solution: No Object Mutation

```
// Replace Step 3 with this code:
john = {
  surname: 'Lennon',
  address: { city: 'Malibu' }
};
```

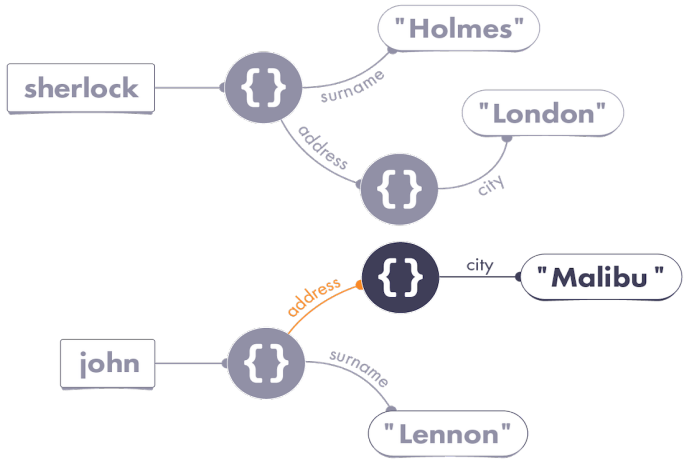
Alternative Solution: No Object Mutation

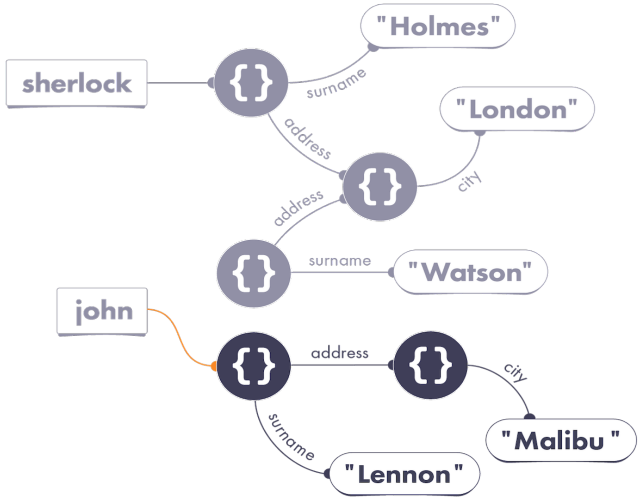
```
// Replace Step 3 with this code:
john = {
  surname: 'Lennon',
  address: { city: 'Malibu' }
};
```



Note that both of these approaches satisfy all of our requirements:

- `console.log(sherlock.surname); // "Holmes"`
- `console.log(sherlock.address.city); // "London"`
- `console.log(john.surname); // "Lennon"`
- `console.log(john.address.city); // "Malibu"`





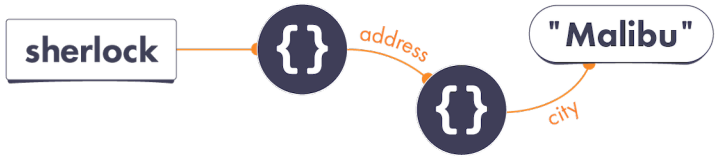
“When you have eliminated the impossible, whatever remains, however improbable, must be the truth.”

-Sherlock Holmes

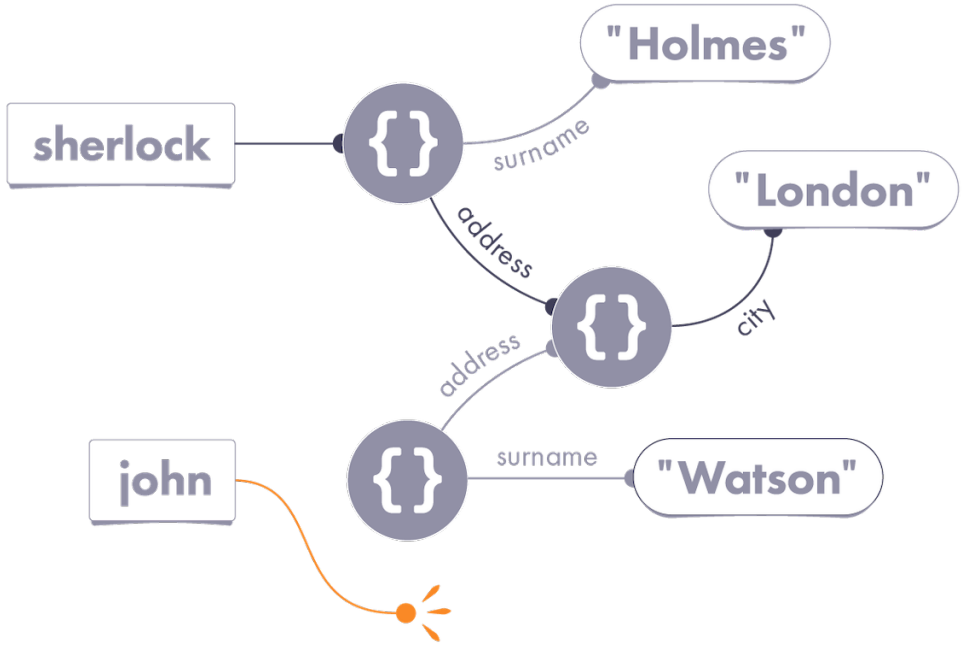
Learn from Sherlock

If you know that `sherlock.address.city` has changed after running some code, the wires from our diagram suggest three explanations:

- Maybe the `sherlock` variable was reassigned.
- Maybe the object we could reach via `sherlock` was mutated, and its `address` property was set to something different.
- Maybe the object we could reach via `sherlock.address` was mutated, and its `city` property was set to something different.



If we point the `john` variable to a different object, we can be fairly sure that `sherlock.address.city` won't change. Our diagram shows that changing the `john` wire doesn't affect any chains starting with `sherlock`:



Let vs Const

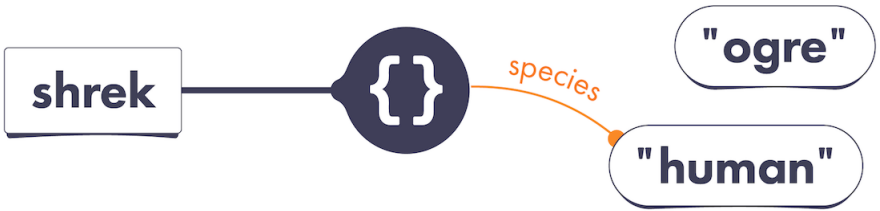
The `const` keyword lets you create read-only variables — also known as *constants*. Once we declare a constant, we can't point it at a different value:

```
const shrek = { species: 'ogre' };
shrek = fiona; // TypeError
```

Let vs Const

We can still mutate the object const points at:

```
const shrek = { species: 'ogre' };
shrek.species = 'human';
console.log(shrek.species); // 'human'
```



Is Mutation Bad?

Is Mutation Bad?

Mutation makes it easy to change some data and immediately “see” the change across the whole program.

However, undisciplined mutation makes it harder to predict what the program would do.

Thing to Remember

Objects are never “nested” in our universe.

Pay close attention to which wire is on the left side of assignment.

Changing an object’s property is also called mutating that object.

Mutating accidentally shared data may cause bugs.

Mutating the objects you’ve just created in code is safe.

Declaring a variable with `const` enforces that this variable’s wire always points at the same value.

`const` does not prevent object mutation!

Exercises

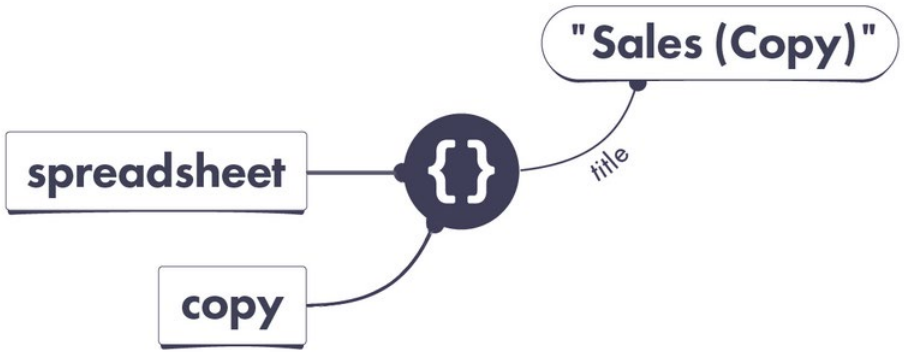
First, **sketch a diagram** of variables and values after this snippet of code runs. Then, **use your diagram** to figure out what the code will print on the last line, and **write the result into the input box at the bottom.**

If this code produces an error, write "error" as an answer.

```
const spreadsheet = { title: 'Sales' };
const copy = spreadsheet;
copy.title = copy.title + ' (Copy)';

console.log(spreadsheet.title); // ???
```

Answer: the last line will print "Sales (Copy)".

We have two variables pointing at the same object: `spreadsheet` and copy. We mutated that object's `title` property. So both `spreadsheet.title` and `copy.title` will give us the updated value.

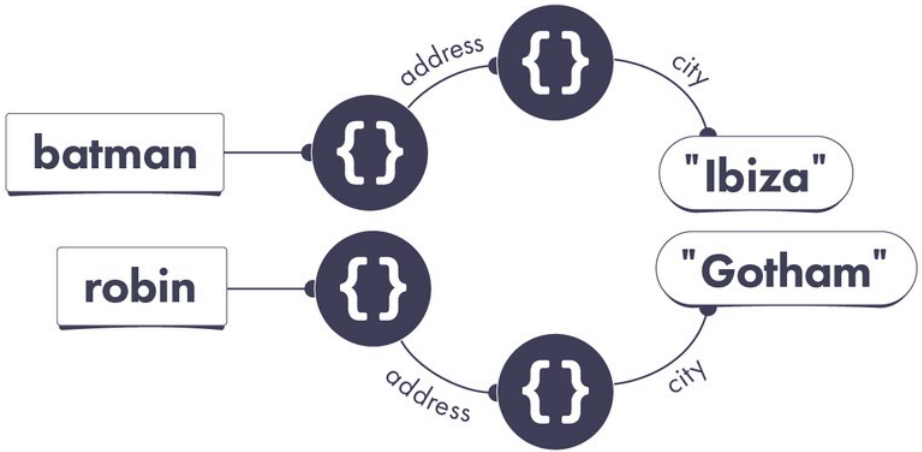


Exercises

First, **sketch a diagram** of variables and values after this snippet of code runs. Then, **use your diagram** to figure out what the code will print on the last line, and **write the result** as an answer.

```
let batman = {
  address: { city: 'Gotham' }
};
let robin = {
  address: batman.address
};
batman.address = { city: 'Ibiza' };

console.log(robin.address.city); // ???
```

Answer: the last line will print "Gotham".

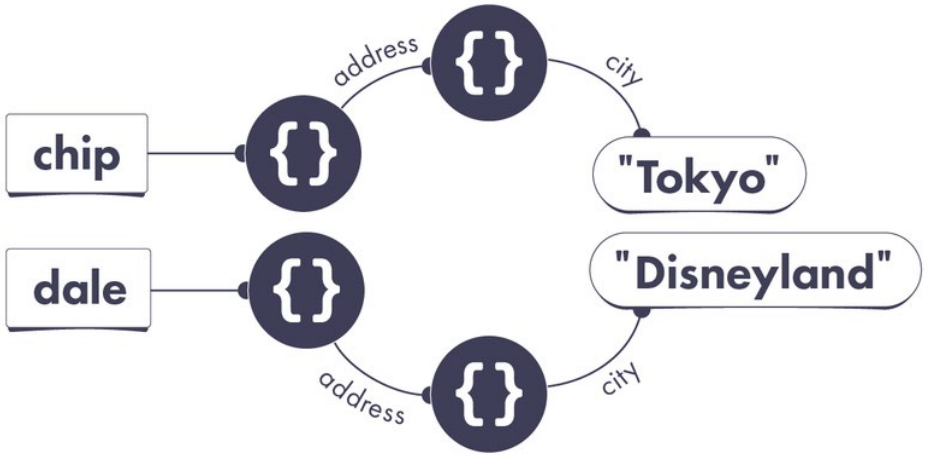


Exercises

First, **sketch a diagram** of variables and values after this snippet of code runs. Then, **use your diagram** to figure out what the code will print on the last line, and **write the result** as an answer.

```
let chip = {
  address: { city: 'Disneyland' }
};
let dale = {
  address: {
    city: chip.address.city
  }
};
chip.address = { city: 'Tokyo' };

console.log(dale.address.city); // ???
```

Answer: the last line will print "Disneyland".

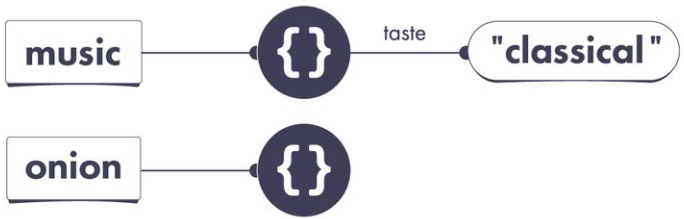


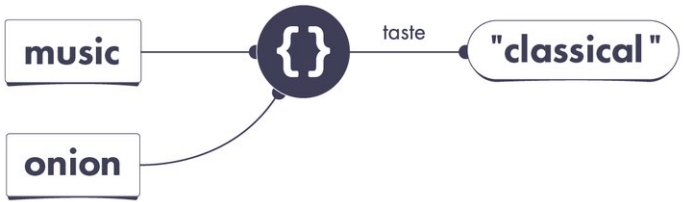
Exercises

Sketch a diagram of variables and values before this snippet of code runs.

Which diagram matches your drawing the best?

```
// ???
console.log(music.taste); // "classical"
onion.taste = 'umami';
console.log(music.taste); // "umami"
```



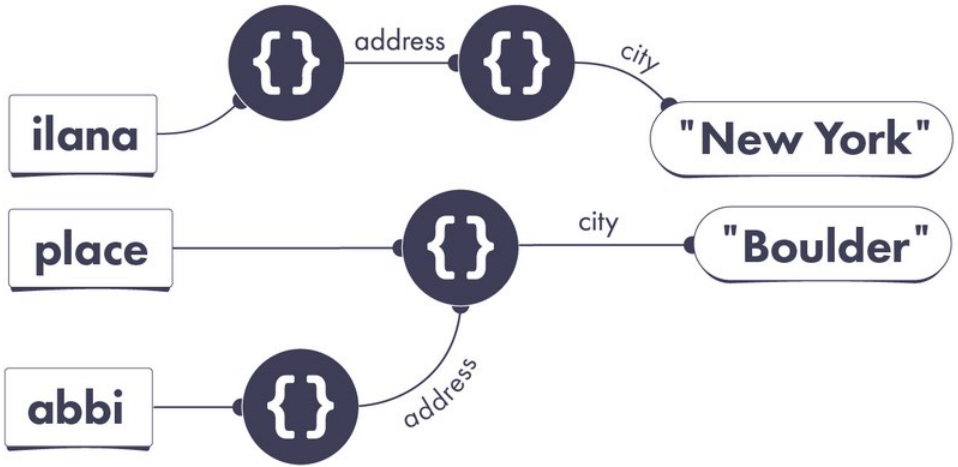


Exercises

First, **sketch a diagram** of variables and values after this snippet of code runs. Then, **use your diagram** to figure out what the code will print on the last line, and **write the result** as an answer.

```
let ilana = {
  address: { city: 'New York' }
};
let place = ilana.address;
place = { city: 'Boulder' };
let abbi = {
  address: place,
};

console.log(ilana.address.city); // ???
```

Answer: the last line will print "New York".

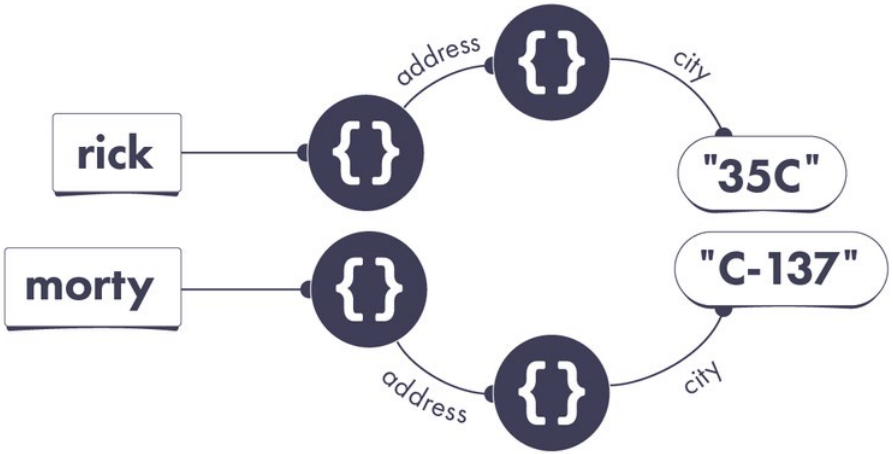


Exercises

First, **sketch a diagram** of variables and values after this snippet of code runs. Then, **use your diagram** to figure out what the code will print on the last line, and **write the result** as an answer.

```
let rick = {
  address: { city: 'C-137' }
};
let morty = {
  address: rick.address
};
rick.address = { city: '35C' };

console.log(morty.address.city); // ???
```

Answer: the last line will print "C-137".



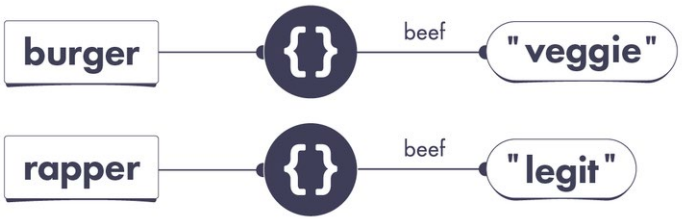
Exercises

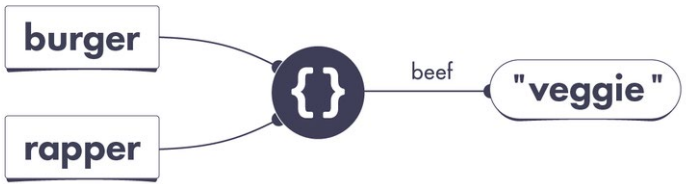
Sketch a diagram of variables and values **before** this snippet of code runs.

```
// ???
console.log(burger.beef); // 'veggie'
burger = rapper;
console.log(burger.beef); // 'legit'
```

Which diagram matches your drawing the most?

```
// ???
console.log(burger.beef); // 'veggie'
burger = rapper;
console.log(burger.beef); // 'legit'
```



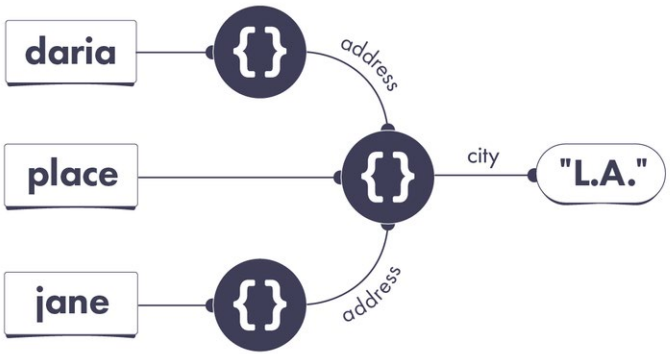


Exercises

Sketch a diagram of variables and values **after** this snippet of code runs.

```
let daria = {
  address: { city: 'Lawndale' }
};
let place = daria.address;
place.city = 'L.A.';
let jane = {
  address: place,
};

console.log(daria.address.city); // ???
```

Answer: the last line will print "L. A."

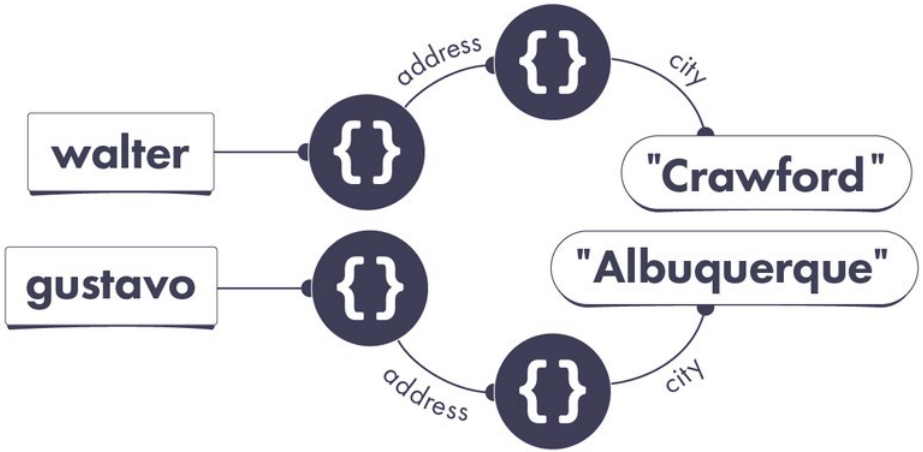


Exercises

Sketch a diagram of variables and values **after** this snippet of code runs.

```
let walter = {
  address: { city: 'Albuquerque' }
};
let gustavo = {
  address: walter.address,
};
walter = {
  address: { city: 'Crawford' }
};

console.log(gustavo.address.city); // ???
```

Answer: the last line will print "Albuquerque"

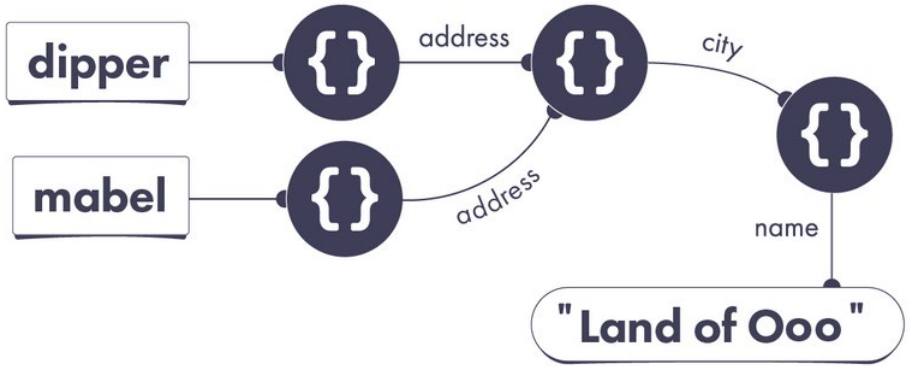


Exercises

Sketch a diagram of variables and values **after** this snippet of code runs.

```
let dipper = {
  address: {
    city: { name: 'Gravity Falls' }
  }
};

let mabel = {
  address: dipper.address
};

dipper.address.city = {
  name: 'Land of Ooo'
};

console.log(mabel.address.city.name); // ???
```

Answer: the last line will print "Land of Ooo"



Exercises

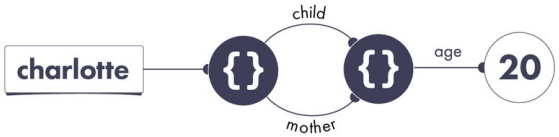
Two of these theories are plausible.

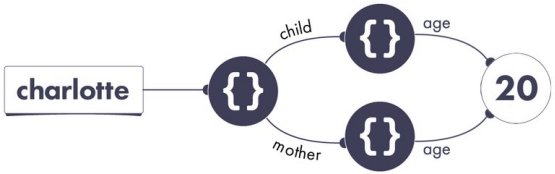
One of them is probably wrong.

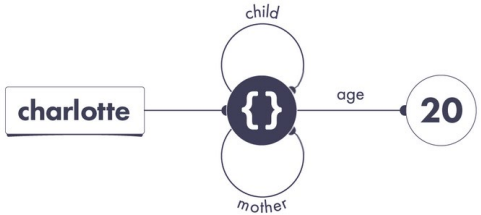
Choose which of these theories is **probably wrong**.

Prepare to explain why, too.

```
// ???
console.log(charlotte.mother.age); // 20
console.log(charlotte.child.age); // 20

charlotte.mother.age = 21;

console.log(charlotte.mother.age); // 21
console.log(charlotte.child.age); // 21
```





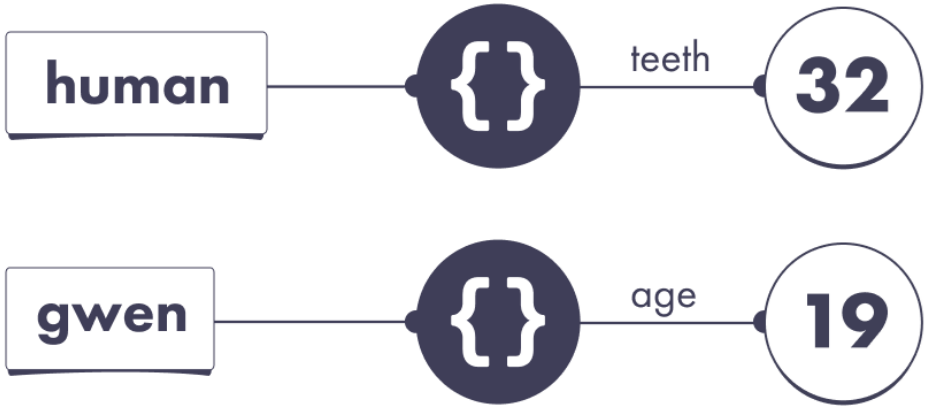


Here is a small riddle to check our mental model:

```
let pizza = {};

console.log(pizza.taste); // "pineapple"
```

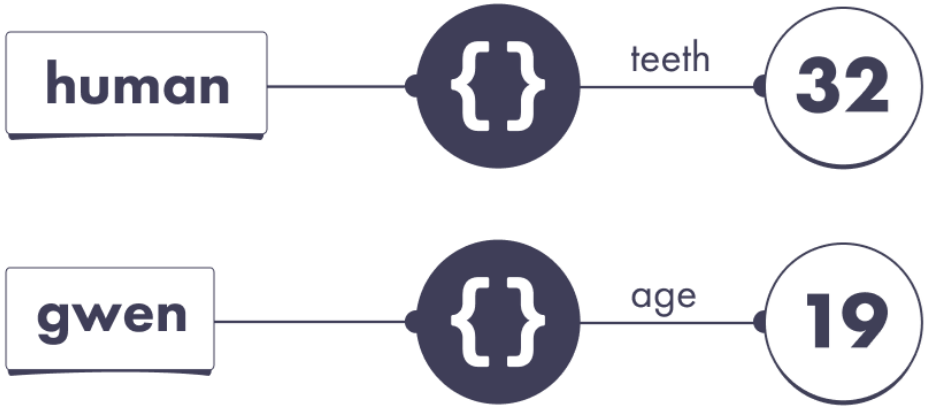
Prototypes

```
let human = {
  teeth: 32
};

let gwen = {
  age: 19
};
```



Prototypes

```
let human = {
  teeth: 32
};

let gwen = {
  age: 19
};

console.log(gwen.teeth); // undefined
```



We can instruct JavaScript to *continue searching* for *our missing property on another object*.

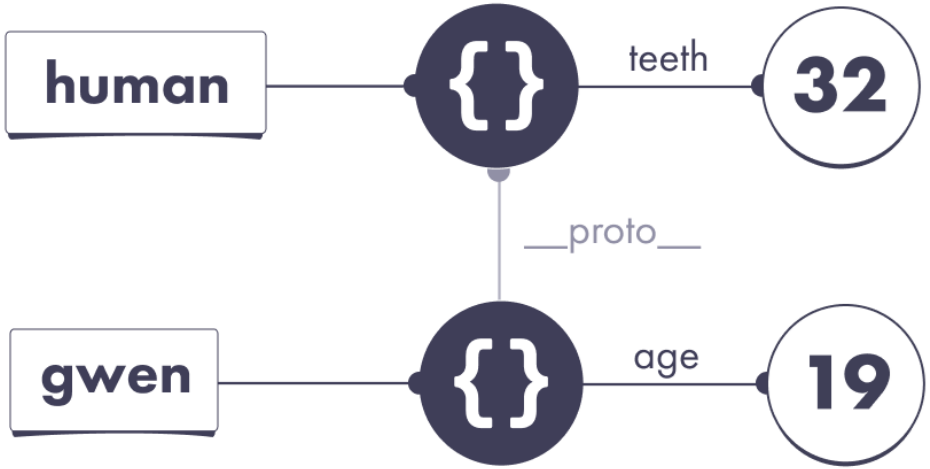
Prototypes

```
let human = {
  teeth: 32
};

let gwen = {
  // We added this line:
  __proto__: human,
  age: 19
};
```

Any JavaScript object may choose another object as a prototype.

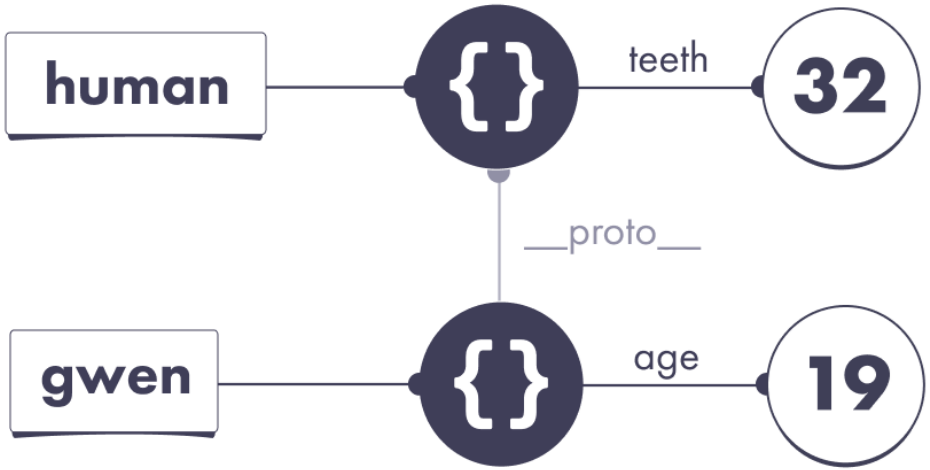
Prototypes

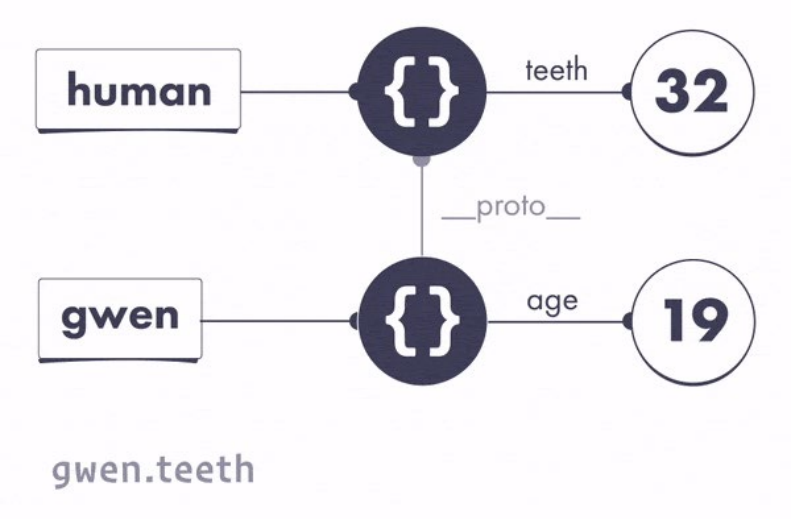
```
let human = {
  teeth: 32
};

let gwen = {
  // We added this line:
  __proto__: human,
  age: 19
};
```



Prototypes

```
let human = {
  teeth: 32
};

let gwen = {
  // We added this line:
  __proto__: human,
  age: 19
};

console.log(gwen.teeth); // 32
```





1. Follow the gwen wire. It leads to an object.

2. Does this object have a teeth property?
    - No.
    - **But it has a prototype.** Let's check it out.

3. Does *that* object have a teeth property?
    - Yes, it points at 32.
    - Therefore, the result of gwen.teeth is 32.

Write down your answers:

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};
```

```
console.log(human.age); // ?

console.log(gwen.age); // ?


console.log(human.teeth); // ?

console.log(gwen.teeth); // ?


console.log(human.tail); // ?

console.log(gwen.tail); // ?
```

Write down your answers:

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};
```

```
console.log(human.age); // undefined

console.log(gwen.age); // 19


console.log(human.teeth); // 32

console.log(gwen.teeth); // 32


console.log(human.tail); // undefined

console.log(gwen.tail); // undefined
```

The Prototype Chain

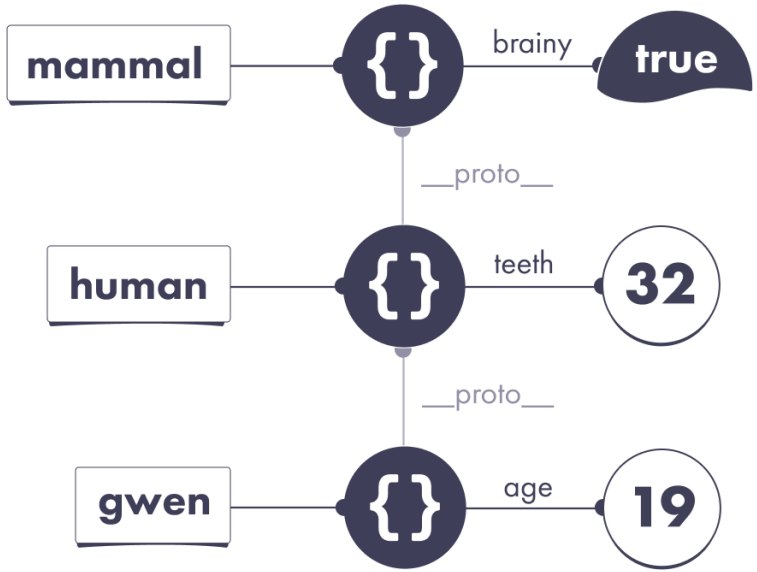
A prototype is more like a relationship. An object may point at another object as its prototype.

The Prototype Chain

```
let mammal = {
  brainy: true,
};

let human = {
  __proto__: mammal,
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};

console.log(gwen.brainy); // true
```

The Prototype Chain

```
let mammal = {
  brainy: true,
};

let human = {
  __proto__: mammal,
  teeth: 32
};

let gwen = {
  __proto__: human,
  age: 19
};

console.log(gwen.brainy); // true
```

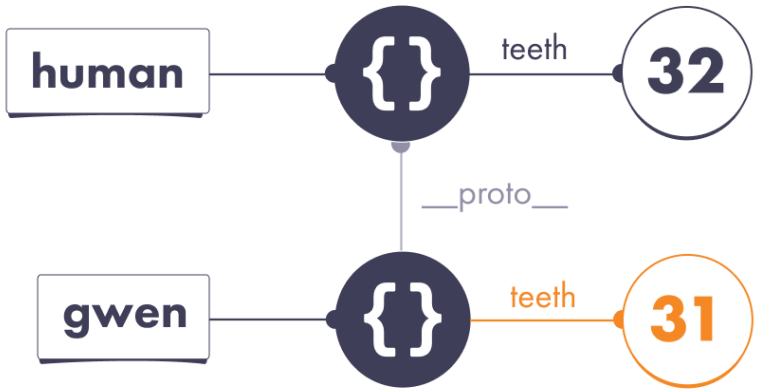


Shadowing

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // This object has its own teeth property:
  teeth: 31
};
```

Shadowing

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // This object has its own teeth property:
  teeth: 31
};
```



gwen.teeth

Shadowing

Once we find our property, we stop the search.

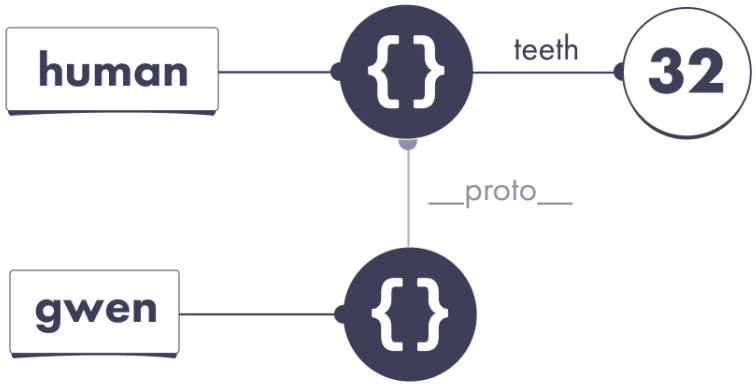
```
// true
console.log(human.hasOwnProperty('teeth'));

// true
console.log(gwen.hasOwnProperty('teeth'));
```

Assignment

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // Note: no own teeth property
};

gwen.teeth = 31;

console.log(human.teeth); // ?
console.log(gwen.teeth); // ?
```

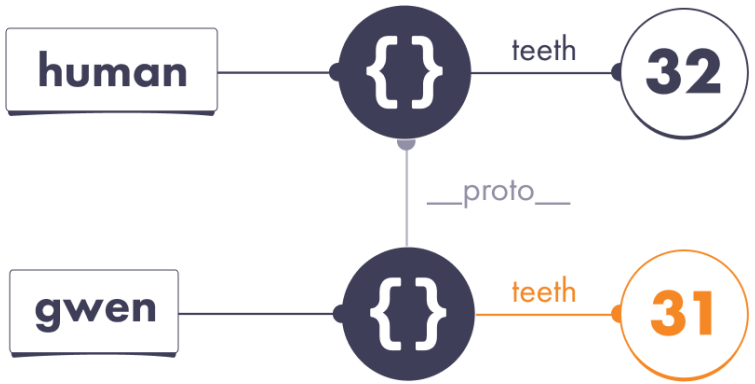
Assignment

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // Note: no own teeth property
};

gwen.teeth = 31;

console.log(human.teeth); // ?
console.log(gwen.teeth); // ?
```



Assignment

```
let human = {
  teeth: 32
};

let gwen = {
  __proto__: human,
  // Note: no own teeth property
};

gwen.teeth = 31;

console.log(human.teeth); // 32
console.log(gwen.teeth); // 31
```



The Object Prototype

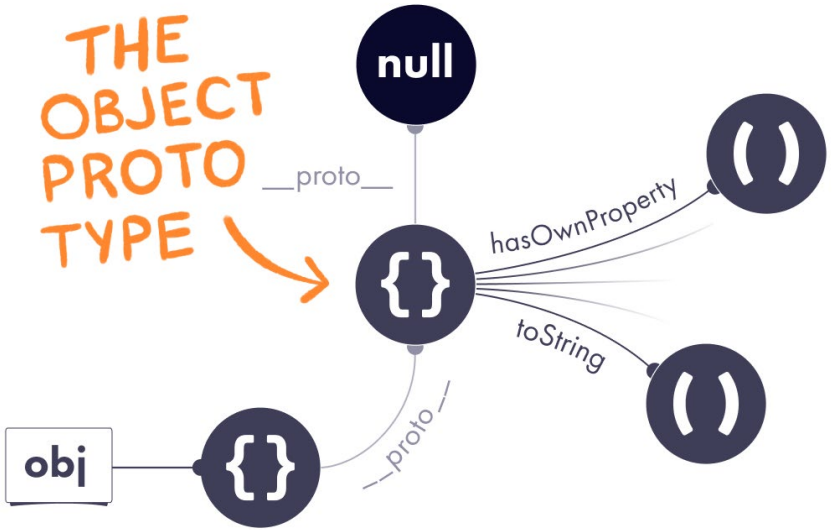
This object doesn't have a prototype, right?

```
let obj = {};

console.log(obj.__proto__); // Play with it!
```

The Object Prototype

This object doesn't have a prototype, right?

```
let obj = {};

console.log(obj.__proto__);
```



The Object Prototype

This explains why the JavaScript objects seem to have “built-in” properties:

```
let human = {
  teeth: 32
};
console.log(human.hasOwnProperty); // function hasOwnProperty() { }
console.log(human.toString); // function toString() { }
```

These “built-in” properties are exist on the Object Prototype.

An Object with No Prototype

Can we set __proto__ to null?

```
let weirdo = {
  __proto__: null
};
```

An Object with No Prototype

Can we set __proto__ to null?

```
let weirdo = {
  __proto__: null
};
```

This will produce an object that truly doesn't have a prototype, at all.

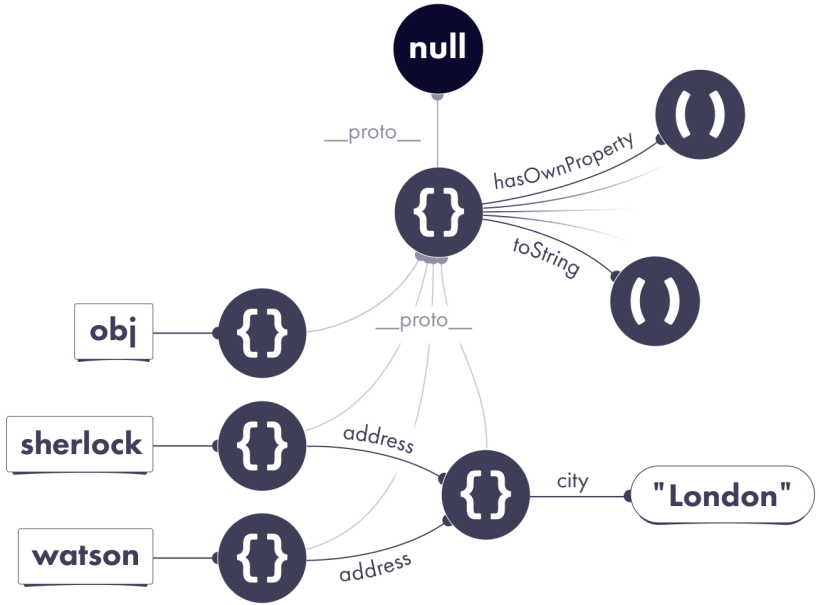
As a result, it doesn't even have built-in object methods:

```
console.log(weirdo.hasOwnProperty); // undefined
console.log(weirdo.toString); // undefined
```

Polluting the Prototype

Let’s illustrate our past example with a new level of detail:

```
let sherlock = {
  address: {
    city: 'London'
  }
};
let watson = {
  address: sherlock.address
};
```



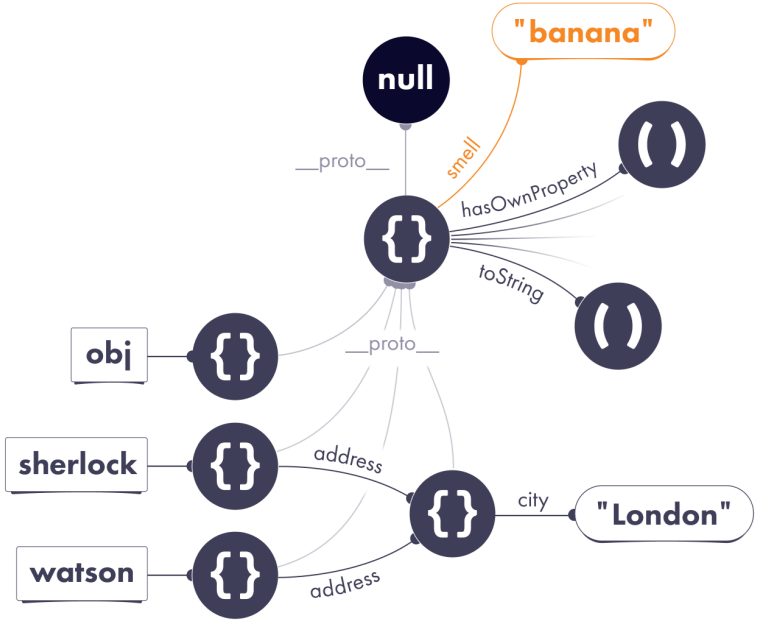
Can we make new properties “appear” on all objects by mutating the prototype?

Polluting the Prototype

```
let sherlock = {
  address: {
    city: 'London'
  }
};
let watson = {
  address: sherlock.address
};
let obj = {};
obj.__proto__.smell = 'banana';

console.log(sherlock.smell); // "banana"
console.log(watson.smell); // "banana"
```

Polluting the Prototype

```
let sherlock = {
  address: {
    city: 'London'
  }
};
let watson = {
  address: sherlock.address
};
let obj = {};
obj.__proto__.smell = 'banana';

console.log(sherlock.smell); // "banana"
console.log(watson.smell); // "banana"
```



Mutating a shared prototype is called *prototype pollution*.

In fact, even using the __proto__ syntax directly itself is discouraged.

```
// Classes.js
class Spiderman {
 lookOut() {
   alert('My Spider-Sense is tingling.');
 }
}

let miles = new Spiderman();
miles.lookOut();
```

```
// Prototype.js
// class Spiderman {
let SpidermanPrototype = {
 lookOut() {
   alert('My Spider-Sense is tingling.');
 }
};

// let miles = new Spiderman();
let miles = { __proto__:
SpidermanPrototype };
miles.lookOut();
```

Thing to Remember

When reading `obj.prop`, if obj doesn't have a prop property, JavaScript will look for `obj.__proto__.prop`.

When writing to `obj.prop`, JavaScript will usually write to the object directly instead of traversing the prototype chain.

We can use `obj.hasOwnProperty('prop')` to determine whether our object has an own property called prop.

We can "*pollute*" a prototype by mutating it.

Exercises

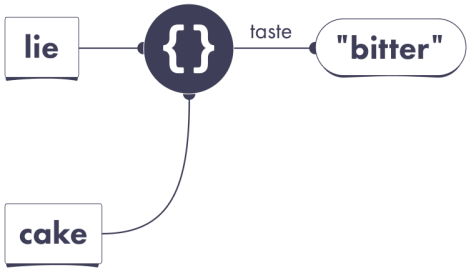
Sketch a diagram of variables and values **after** this snippet of code runs.

```
let lie = {
  taste: 'bitter'
};

let cake = {
  __proto__: lie
};
```

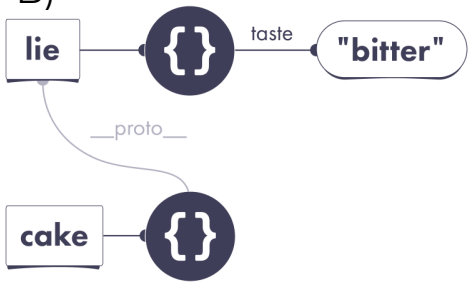
Which diagram matches your drawing the best?

```
let lie = {
  taste: 'bitter'
};

let cake = {
  __proto__: lie
};
```

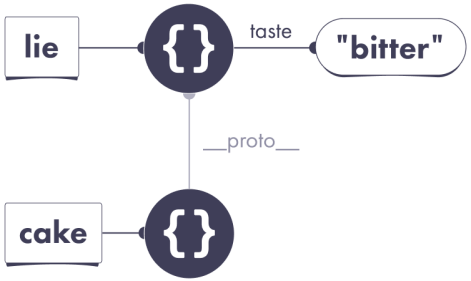
A)



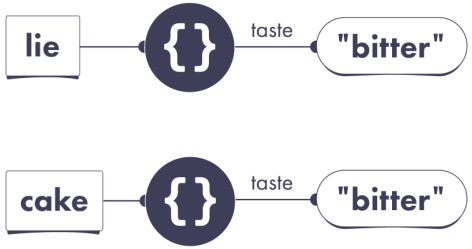
B)



C)



D)

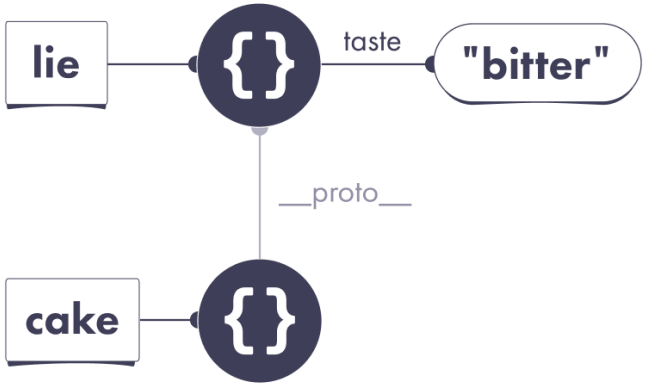


Use this diagram to answer these three questions:

1. console.log(cake === lie)

2. console.log(cake.taste === lie.taste)

3. cake.hasOwnProperty('taste') === lie.hasOwnProperty('taste')



Exercises

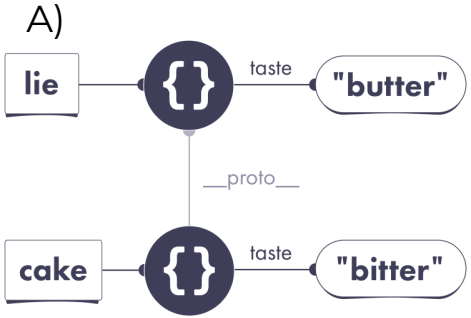
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

```
let lie = {
  taste: 'bitter'
};

let cake = {
  __proto__: lie
};

lie.taste = 'butter';
```

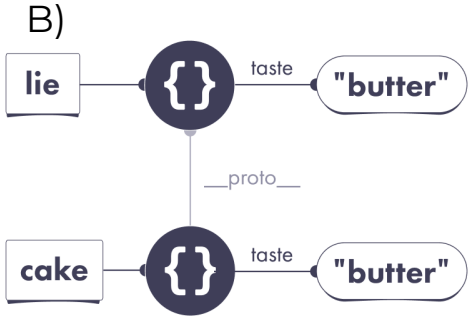
Which diagram matches your drawing the most?

```
let lie = {
  taste: 'bitter'
};

let cake = {
  __proto__: lie
};

lie.taste = 'butter';
```

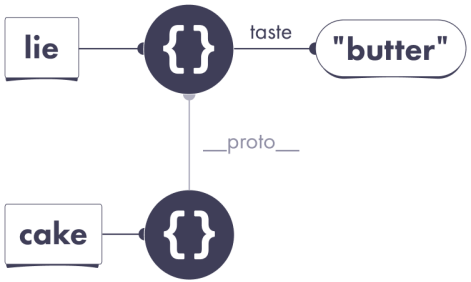
A)



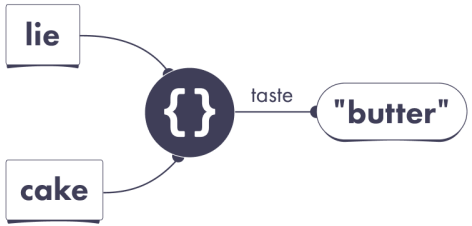
B)



C)



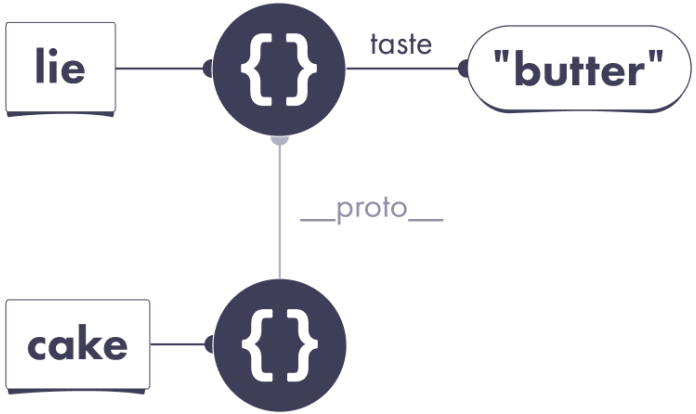
D)



Use this diagram to answer these two questions:

1. console.log(lie.taste)

2. console.log(cake.taste)



Exercises

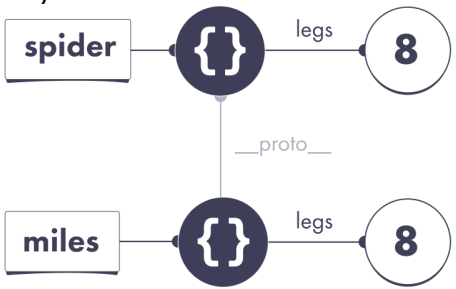
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.

```
let spider = {
  legs: 8
};
let miles = {
  __proto__: spider
};

miles.legs = 2;
```

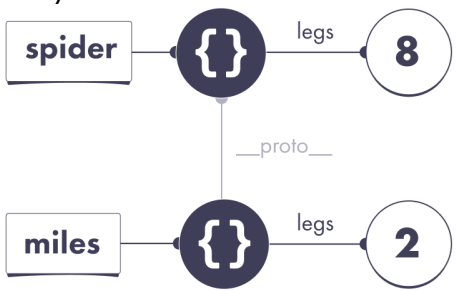
Which diagram matches your drawing the most?

```
let spider = {
  legs: 8
};
let miles = {
  __proto__: spider
};

miles.legs = 2;
```

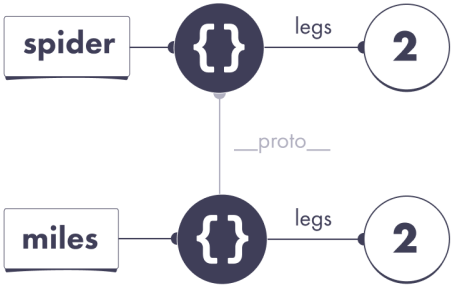
A)



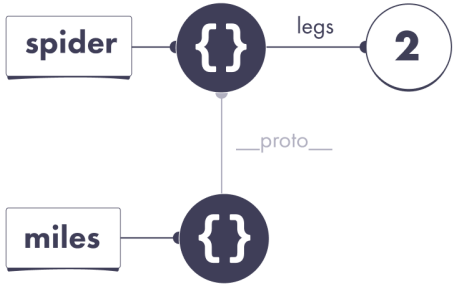
B)



C)



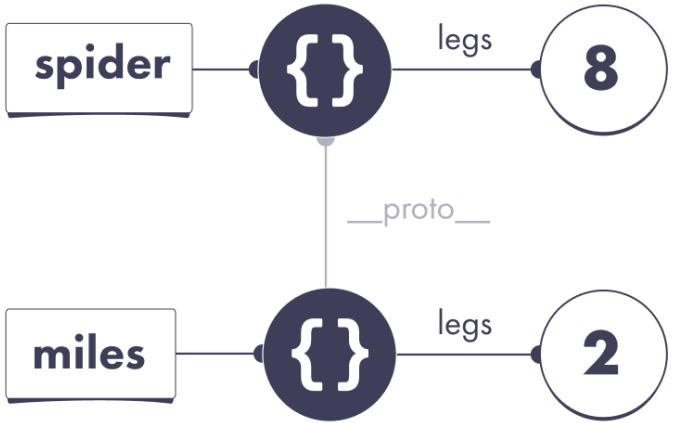
D)



Use this diagram to answer these two questions:

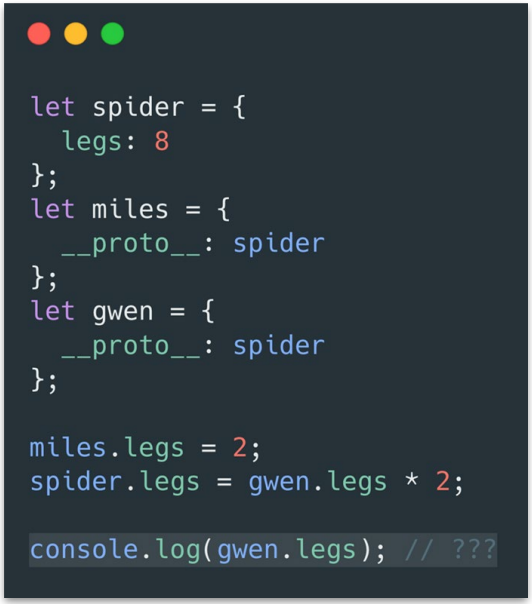
1. console.log(spider.legs)

2. console.log(miles.legs)



Exercises

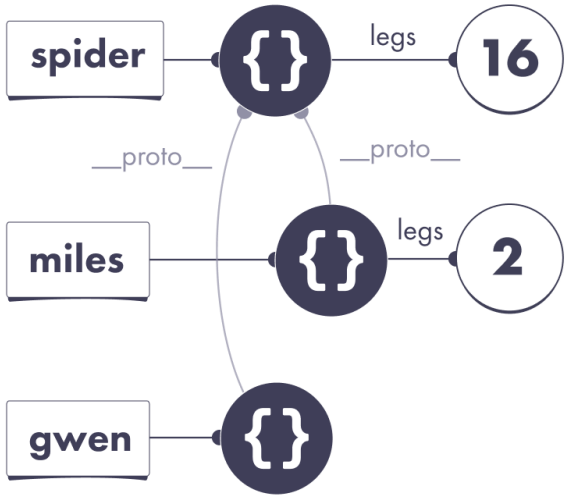
Sketch a diagram of variables and values after this snippet of code runs. Use our mental model.



```
let spider = {
  legs: 8
};
let miles = {
  __proto__: spider
};
let gwen = {
  __proto__: spider
};

miles.legs = 2;
spider.legs = gwen.legs * 2;

console.log(gwen.legs); // ???
```

Answer: 16 is the correct answer.

The object that gwen points at doesn't have a legs property, so we continue the search on its prototype. We find the legs property there, pointing at 16. That's our answer.



The second line of this code is a mystery. You have two tasks:

1. Draw the universe right after the second line
2. Figure out how the second line really ends

After you're finished, **write the ??? part in the answer field.**

```
let goose = { location: 'heaven' }
let cheese = // ???

// >>> Diagram this moment! <<<

console.log(cheese === goose); // false
console.log(cheese.location); // "heaven"

goose.location = 'hell';
console.log(cheese.location); // "hell"
```