Closure

- We know that a function can access variables outside of it
  - This feature is used quite often in JavaScript
- But what happens when an outer variable changes? Does a function get the most recent value or the one that existed when the function was created?

```
let name = "John";

function sayHi() {
    alert("Hi, " + name);
}

name = "Adam";

sayHi(); // what will it show: "John" or "Adam"?
```

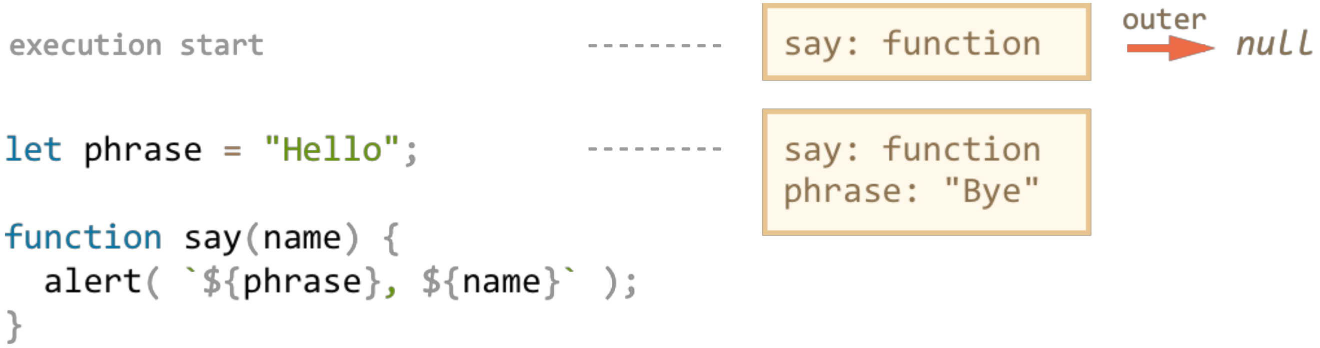
Lexical Environment

- In JavaScript, every running function, code block, and the script as a whole have an associated object known as the *Lexical Environment*
- The **Lexical Environment** object consists of two parts:
  - *Environment Record* – an object that has all local variables as its properties (and some other information like the value of this)
  - A reference to the *outer lexical environment*, usually the one associated with the code lexically right outside of it (outside of the current curly brackets)
- So, a “variable” is just a property of the special internal object, Environment Record
  - “To get or change a variable” means “to get or change a property of the Lexical Environment”
- For instance, in this simple code, there is only one Lexical Environment
  - the global environment associated with the whole script

```
let phrase = "Hello";

alert(phrase);
```

Function Declaration

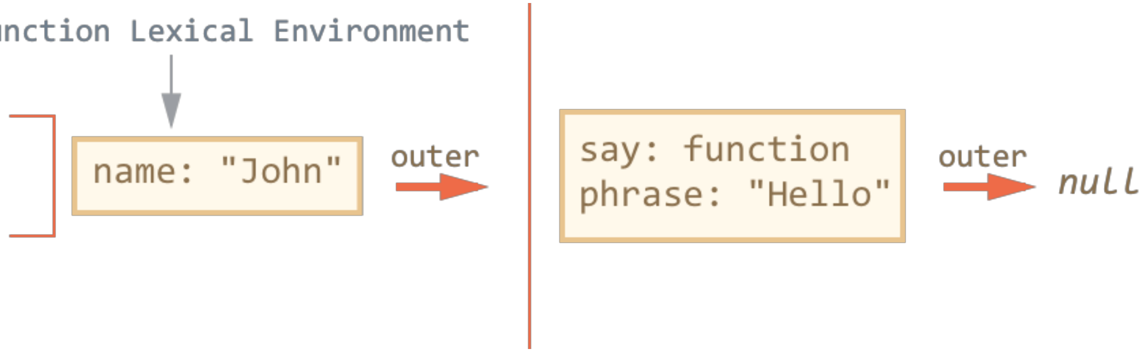
- Function declarations are special
- Unlike let variables, they are processed not when the execution reaches them, but when a Lexical Environment is created
  - For the global Lexical Environment, it means the moment when the script is started
- That is why we can call a function declaration before it is defined



Inner and Outer Lexical Environments

- When a function runs, a new function Lexical Environment is created automatically
- That Lexical Environment is used to store local variables and parameters of the call
- Here’s the picture of Lexical Environments when the execution is inside say("John"), at the line labeled with an arrow:

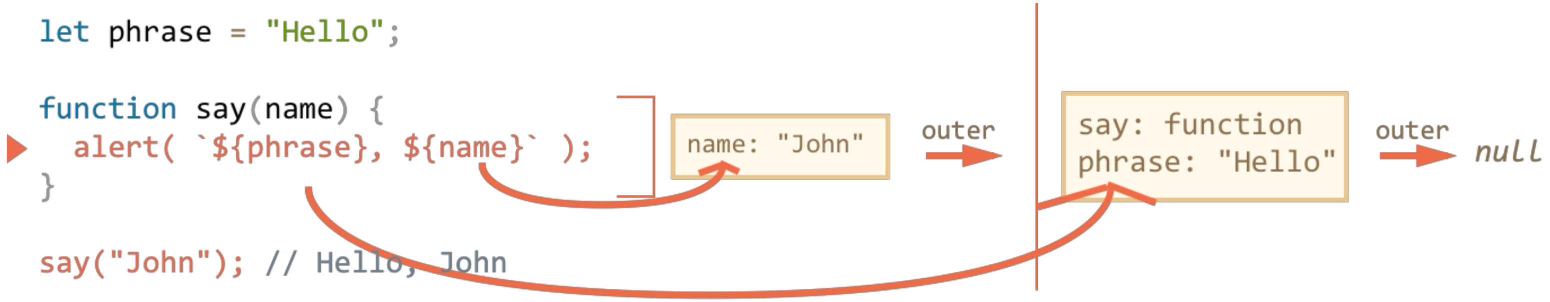
```
let phrase = "Hello";

function say(name) {
  alert( `${phrase}, ${name}` );
}

say("John"); // Hello, John
```



- During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global)
  - The inner Lexical Environment has the outer reference to the outer one

Inner and Outer Lexical Environments

- When code wants to access a variable – it is first searched for in the inner Lexical Environment, then in the outer one, then the more outer one and so on until the end of the chain



- If a function is called multiple times, then each invocation will have its own Lexical Environment, with local variables and parameters specific for that very run.

Inner and Outer Lexical Environments

- Because of the described mechanism **a function gets outer variables as they are now**
  - It takes its current value from its own or an outer Lexical Environment

```
let name = "John";

function sayHi() {
    alert("Hi, " + name);
}

name = "Adam"; // (*)

sayHi(); // Adam
```

  - The global Lexical Environment has name: "John"
  - At the line (*) the global variable is changed, now it has name: "Adam"
  - When the function say(), is executed and takes name from outside. Here that’s from the global Lexical Environment where it’s already "Adam"

Nested Functions

- A function is called “nested” when it is created inside another function
- We can use it to organize our code, like this:

```
function sayHiBye(firstName, lastName) {

    // helper nested function to use below
    function getFullName() {
        return firstName + " " + lastName;
    }

    alert("Hello, " + getFullName());
    alert("Bye, " + getFullName());
}
```

  - The nested function getFullName() can access the outer variables firstName and lastName

Nested Functions

- A nested function can be returned and then be used somewhere else
- No matter where, it still has access to the same outer variables

```
function makeCounter() {
    let count = 0;

    return function () {
        return count++; // has access to the outer counter
    };
}

let counter = makeCounter();

alert(counter()); // 0
alert(counter()); // 1
alert(counter()); // 2
```

Nested Functions

- When the inner function runs, the variable in count++ is searched from inside out:
  - The locals of the nested function…
  - The variables of the outer function…
  - And so on until it reaches global variables.

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}
```

- In this example count is found on step 2, so count++ finds the outer variable and increases it in the Lexical Environment where it belongs

Nested Functions

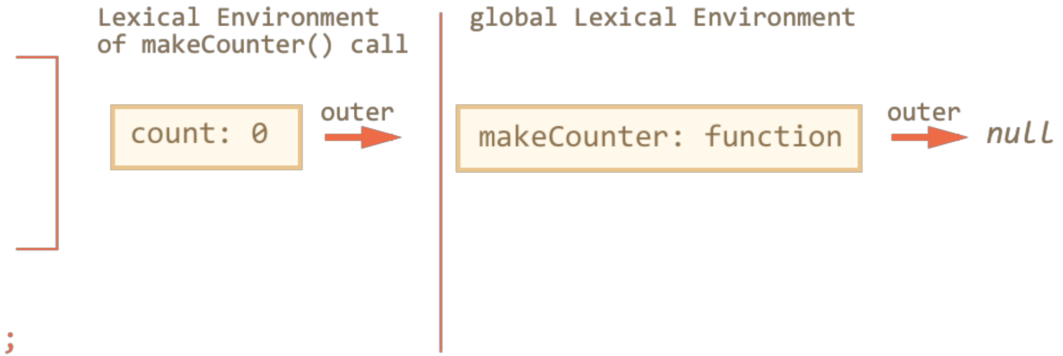
- For every call to makeCounter() a new function Lexical Environment is created, with its own counter
- So the resulting counter functions are independent

```
function makeCounter() {
    let count = 0;
    return function () {
        return count++;
    };
}

let counter1 = makeCounter();
let counter2 = makeCounter();

alert(counter1()); // 0
alert(counter1()); // 1
alert(counter1()); // 2

alert(counter2()); // 0 (independent)
```

Nested Functions

- Behind the scenes, all functions "on birth" receive a hidden property [[Environment]] with a reference to the Lexical Environment of their creation
- At the moment of the call of makeCounter(), the Lexical Environment is created, to hold its variables and argument

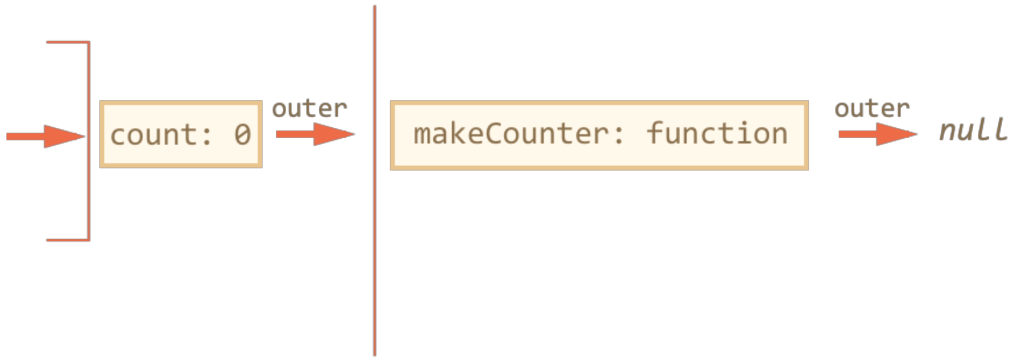
```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();
```



Nested Functions

- During the execution of makeCounter(), a tiny nested function is created
- The value of its [[Environment]] is the current Lexical Environment of make Counter() (where it was born):

```
function makeCounter() {
  let count = 0;

  return function() { [[Environment]]
    return count++;
  };
}

let counter = makeCounter();
```

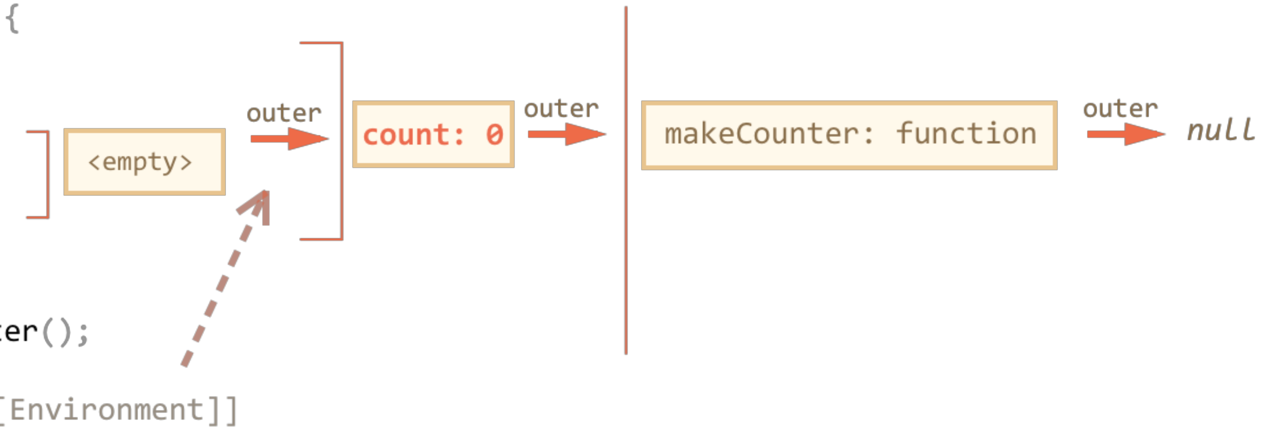


- The result (the tiny nested function) is assigned to the global variable counter

Nested Functions

- When the counter() is called, an “empty” Lexical Environment is created for it
  - It has no local variables by itself.
- But the [[Environment]] of counter is used as the outer reference for it, so it has access to the variables of the former makeCounter() call where it was created:

```
function makeCounter() {
  let count = 0;

  return function() {
    return count++;
  };
}

let counter = makeCounter();

alert( counter() );
```



- When it looks for count, it finds it among the variables makeCounter, in the nearest outer Lexical Environment

Closures

- There is a general programming term “closure”, that developers should know
- A closure is a function that remembers its outer variables and can access them
- In some languages, that’s not possible, or a function should be written in a special way to make it happen
- As explained above, in JavaScript all functions are naturally closures
  - That is: they automatically remember where they were created, and all of them can access outer variables

Code Blocks

- We also can use a “bare” code block {…} to isolate variables into a “local scope”
- For instance, in a web browser all scripts share the same global area
- So if we create a global variable in one script, it becomes available to others
  - That becomes a source of conflicts if two scripts use the same variable name
- If we’d like to avoid that, we can use a code block to isolate the script:

```
{
    // do some job with local variables that should not be seen outside
    let message = "Hello";
    alert(message); // Hello
}

alert(message); // Error: message is not defined
```

  - The code outside of the block (or inside another script) doesn’t see variables inside the block, because the block has its own Lexical Environment

IIFE

- In old scripts, one can find “immediately-invoked function expressions” (IIFE) used for the same purpose
- They look like this:

```
(function () {
    let message = "Hello";
    alert(message); // Hello
})();
```

- Here a Function Expression is created and immediately called
- So the code executes right away and has its own private variables

Garbage Collection

- A Lexical Environment object dies when it becomes unreachable: when no nested functions remain that reference it
- In the code below, after **g** becomes unreachable, **value** is also cleaned from memory:

```
function f() {
    let value = 123;

    function g() { alert(value); }

    return g;
}

let g = f(); // while g is alive its corresponding Lexical Environment lives

g = null; // ...and now the memory is cleaned up
```

- JavaScript engines try to optimize that. They analyze variable usage and if it’s easy to see that an outer variable is not used – it is removed.

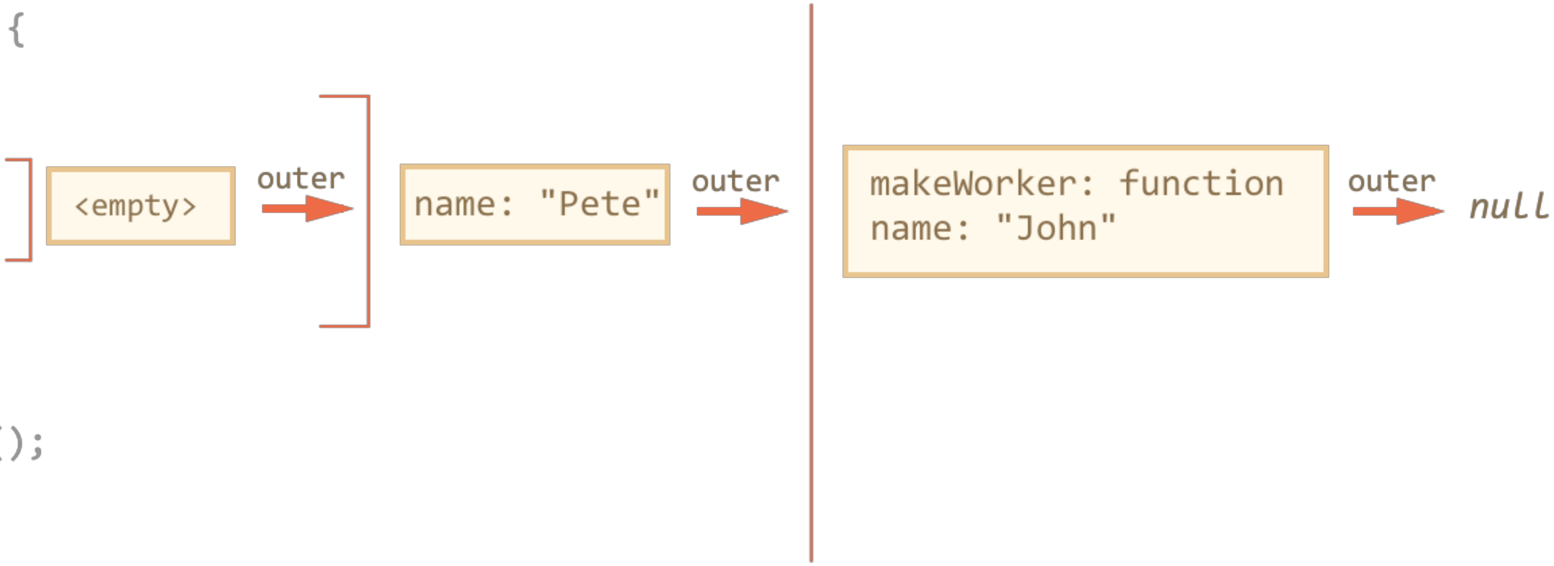
Exercise (24)

- What will be the output of the following function?
- Draw a diagram of the lexical environments when execution reaches the line with (*)

```
function makeWorker() {
    let name = "Pete";

    return function () {
        alert(name); // (*)
    };
}

let name = "John";

// create a function
let work = makeWorker();

// call it
work(); // what will it show? "Pete" (name where created) or "John" (name where called)?
```

Solution

```
function makeWorker() {
  let name = "Pete";

  return function() {
    alert(name);
  };
}

let name = "John";
let work = makeWorker();

work(); // Pete
```



Exercise (25)

- Here a counter object is made with the help of the constructor function
- Will it work? What will it show?

```
function Counter() {
    let count = 0;

    this.up = function () {
        return ++count;
    };
    this.down = function () {
        return --count;
    };
}

let counter = new Counter();

alert(counter.up()); // ?
alert(counter.up()); // ?
alert(counter.down()); // ?
```

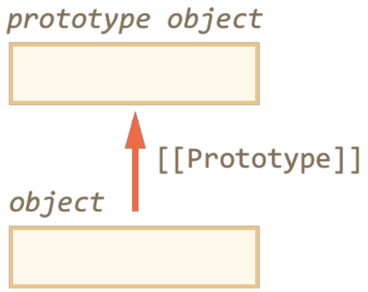
Exercise (26)

- We have a built-in method arr.filter(f) for arrays
  - It filters all elements through the function f. If f returns true, then that element is returned in the resulting array.
- Make a set of “ready to use” filters:
  - inBetween(a, b) – between a and b or equal to them (inclusively)
  - inArray([...]) – in the given array
- For instance:

```
/* .. your code for inBetween and inArray */
let arr = [1, 2, 3, 4, 5, 6, 7];

alert(arr.filter(inBetween(3, 6))); // 3,4,5,6

alert(arr.filter(inArray([1, 2, 10]))); // 1,2
```

Prototype

- In JavaScript, objects have a special hidden property [[Prototype]], that is either null or references another object which is called prototype
- When we want to read a property from object, and it's missing, JavaScript automatically takes it from the prototype
  - This is called "prototypal inheritance"



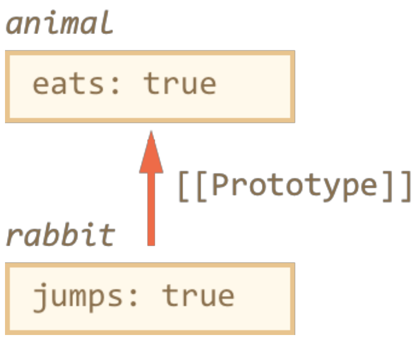
- The property [[Prototype]] is internal and hidden, but there are many ways to set it
- One of them is to use __proto__, like this:

```
let animal = {
    eats: true
};
let rabbit = {
    jumps: true
};

rabbit.__proto__ = animal;
```

Prototype

- If we look for a property in rabbit, and it's missing, JavaScript automatically takes it from animal:

```
// we can find both properties in rabbit now:
alert(rabbit.eats); // true
alert(rabbit.jumps); // true
```

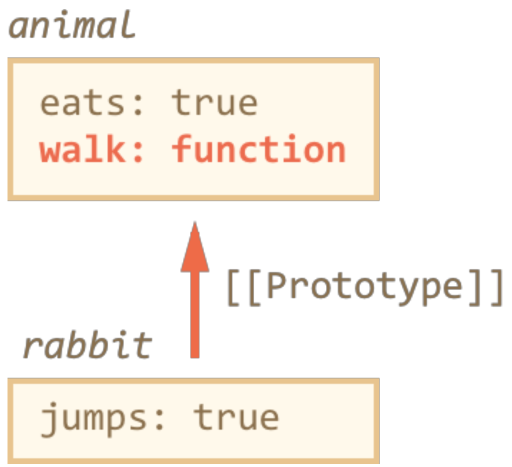


- We say that "animal is the prototype of rabbit"
- So if animal has a lot of useful properties and methods, then they become automatically available in rabbit
- Such properties are called "inherited"

Prototype

- If we have a method in animal, it can be called on rabbit:

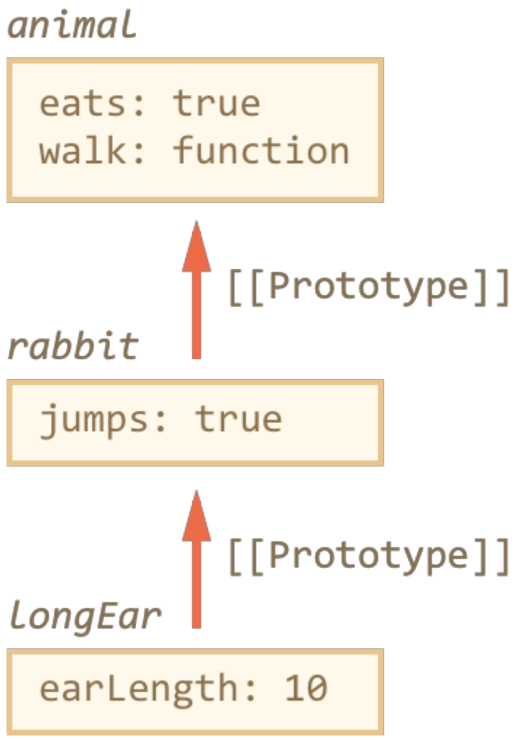
```
let animal = {
    eats: true,
    walk() {
        alert("Animal walk");
    }
};
let rabbit = {
    jumps: true
};

rabbit.__proto__ = animal;

// walk is taken from the prototype
rabbit.walk(); // Animal walk
```



Prototype

- The prototype chain can be longer:

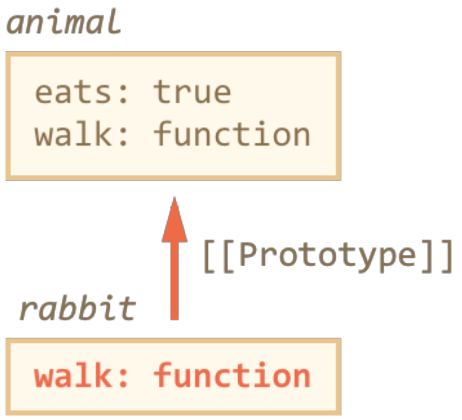
```
let animal = {
    eats: true,
    walk() {
        alert("Animal walk");
    }
};

let rabbit = {
    jumps: true,
    __proto__: animal
};

let longEar = {
    earLength: 10,
    __proto__: rabbit
}

// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```



Read/Write Rules

- The prototype is only used for reading properties
- Write/delete operations work directly with the object
- In the example below, we assign its own walk method to rabbit
  - From that point, rabbit.walk() call finds the method immediately in the object and executes it, without using the prototype

```
let animal = {
    eats: true,
    walk() {  /* this method won't be used by rabbit */
    }
};
let rabbit = {
    __proto__: animal
}

rabbit.walk = function () {
    alert("Rabbit! Bounce-bounce!");
};
rabbit.walk(); // Rabbit! Bounce-bounce!
```

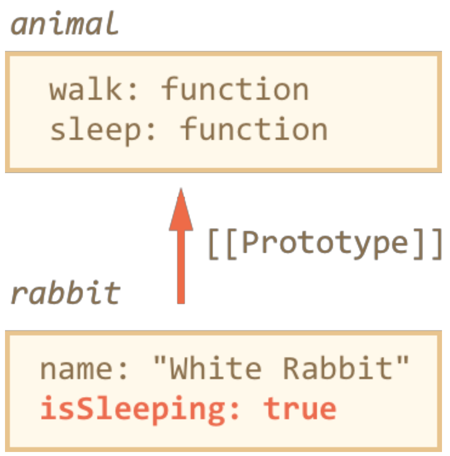


The value of “this”

- If we call obj.method(), and the method is taken from the prototype, ”this” still references obj
- So methods always work with the current object even if they are inherited
- In the example below, the call rabbit.sleep() sets this.isSleeping on the rabbit object:

```
let animal = {
    walk() {
        if (!this.isSleeping) {
            alert('I walk');
        }
    },
    sleep() {
        this.isSleeping = true;
    }
};

let rabbit = {
    name: "White Rabbit",
    __proto__: animal
};
```

```
// modifies rabbit.isSleeping
rabbit.sleep();

alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined
(no such property in the prototype)
```



Exercise (27)

- We have two hamsters: speedy and lazy inheriting from the general hamster object
- When we feed one of them, the other one is also full. Why? How to fix it?

```
let hamster = {
    stomach: [],
    eat(food) {
        this.stomach.push(food);
    }
};
let speedy = {
    __proto__: hamster
};
let lazy = {
    __proto__: hamster
};

// This one found the food
speedy.eat("apple");
alert(speedy.stomach); // apple
// This one also has it, why? fix please.
alert(lazy.stomach); // apple
```