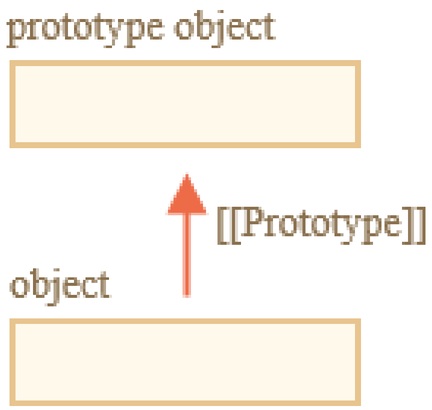
[[Prototype]]

- every object has special hidden property [[Prototype]]
  - either null or references another object.
  - object is called “a prototype”:

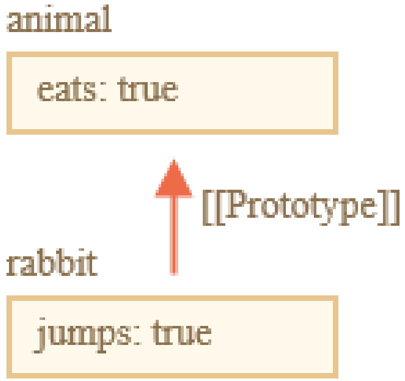


- read a property from object, and it’s missing,
  - JavaScript automatically takes it from the prototype.
  - called “prototypal inheritance”.
  - property [[Prototype]] is internal and hidden, but there are many ways to set it.

Inherit properties

- If look for a property in rabbit, and it’s missing, JavaScript automatically takes it from animal.

- console.log tries to read property rabbit.eats (**),
  - JavaScript follows the [[Prototype]] reference and finds it in animal

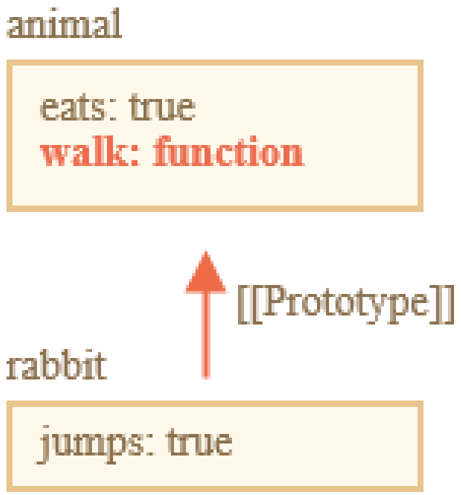


```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};
rabbit.__proto__ = animal; // (*) __proto__ is a ‘sneaky’ (deprecated) way to access [[Prototype]]

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

Inherit methods

- method in animal, it can be called on rabbit

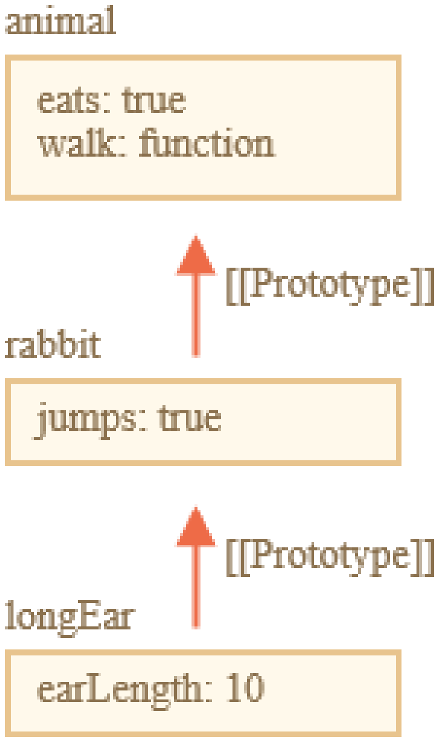
```
let animal = {
  eats: true,
  walk:  function() {
  console.log("Animal walk");
  }
};

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

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



Prototype chain

- prototype chain can be longer
- restrictions:
  - references can’t go in circles..
  - value of __proto__ can be either an object or null.
  - there can be only one [[Prototype]]. An object may not inherit from two others.

```
let animal = {
  eats: true,
  walk: function() {
  console.log("Animal walk");
  }
};
let rabbit = {
  jumps: true,
  __proto__: animal
};
let longEar = {
  earLength: 10,
  __proto__: rabbit
};
```



Own properties do not use prototype chain

- Properties declared on an object work directly with the object
  - “shadow”/override anything further up the prototype chain

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

let rabbit = {
  __proto__: animal
};

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

- From now on, rabbit.walk() call finds the method in the object without using prototype

```
rabbit.walk(); // Rabbit! Bounce-bounce!
```

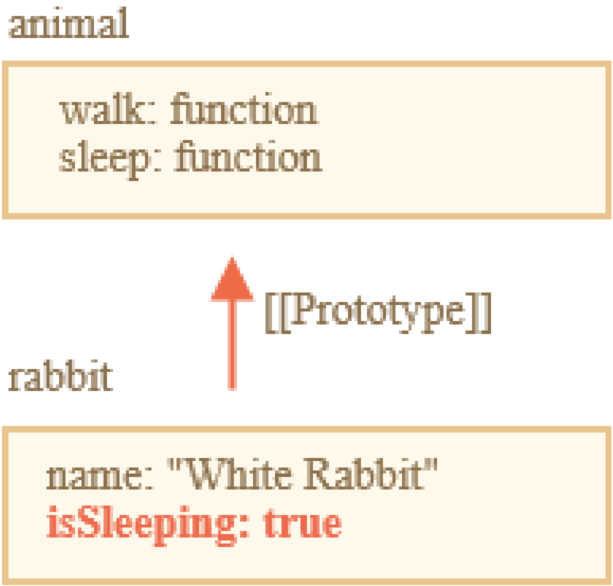
The value of “this”

- what’s the value of this inside an inherited method
  - answer: this is not affected by prototypes at all.
  - No matter where the method is found:
    - in an object or its prototype
    - this is always the object before the dot
- a super-important thing,
  - may have a big object with many methods and inherit from it.
  - descendent objects can run its methods, and they will modify their own state
- methods are often shared, but the object state generally is not

methods often shared, object state generally not

*

```
// animal has methods
let animal = {
  walk: function() {
    if (!this.isSleeping) {
      alert(`I walk`);
    }
  },
  sleep: function() {
    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

1. Use __proto__ so any property lookup will follow the path: pockets → bed → table → head.

pockets.pen should be 3

bed.glasses should be 1

2. Draw the object diagram with objects and labeled arrows for the [[Prototype]] links

```
let head = {
    glasses: 1
};

let table = {
    pen: 3
};

let bed = {
    sheet: 1,
    pillow: 2
};

let pockets = {
    money: 2000
};

console.log("expect 3: ", pockets.pen);
console.log("expect 1: ", bed.glasses);
```

For…in loop

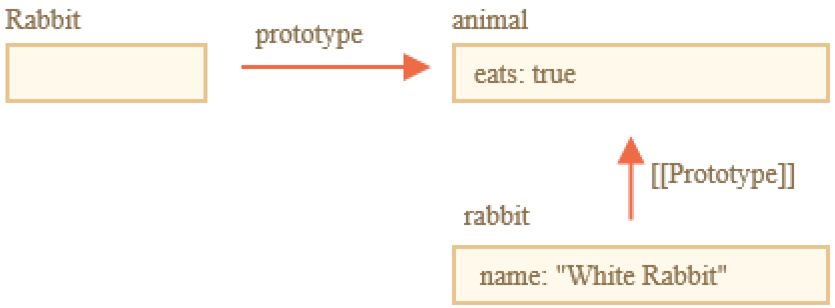
- for..in loops over inherited properties too.

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

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

// Object.keys only return own keys
console.log(Object.keys(rabbit)); // jumps

// for..in loops over both own and inherited keys
for(let prop in rabbit) console.log(prop); // jumps, then eats
```

F.prototype -- Set [[Prototype]] using constructor function

- If MyConstructor.prototype is an object,
  - new operator sets it to [[Prototype]] for the new object.
- MyConstructor.prototype is a regular property named "prototype"
  - This is not the ‘special hidden’ [[Prototype]] property
  - regular property with this name
- When ‘new’ is called sets [[Prototype]] to MyConstructor.prototype

```
let animal = {
  eats: true
};
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype = animal;
let rabbit = new Rabbit("White Rabbit"); //rabbit.__proto__ == animal
 console.log( rabbit.eats ); // true
```



Class syntax

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }  ...
}  //no comma between methods (not an object literal)
```

Then use new MyClass() to create a new object with all the listed methods.

The constructor() method is called automatically by new, so we can initialize the object there.

Class syntax

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}

// Usage:
let user = new User("John");
user.sayHi();
```

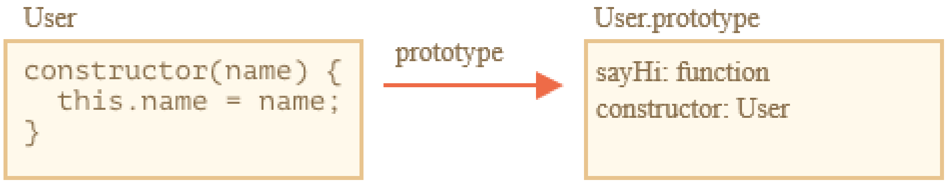
- When new User("John") is called:
  - A new object is created.
  - The constructor runs with the given argument and assigns it to this.name
  - …Then we can call object methods, such as user.sayHi().

JavaScript classes are (constructor) functions

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}
// proof: User is a function
console.log(typeof User); // function

// Usage:
let user = new User("John");
user.sayHi();
```

- Creates a constructor function named User,
  - result of the class declaration.
  - constructor function code taken from the constructor method
    - assumed empty if we don’t write such method).
  - Stores class methods, such as sayHi, in User.prototype.

- Afterwards, for new User objects,
  - call a method, it’s taken from the prototype
  - object has access to class methods.



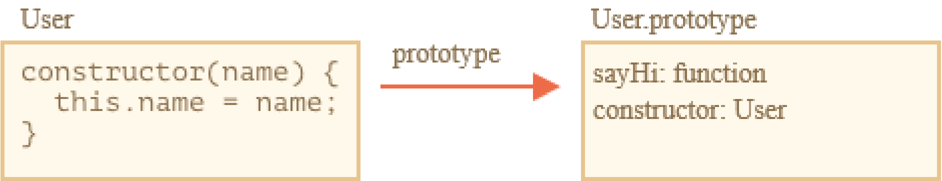
```
constructor(name) {
  this.name = name;
}
```

Could write using just constructor function

```
function User(name) {
  this.name = name;
}

User.prototype.sayHi = function() {
  alert(this.name);
};
```

```
// Usage
let user = new User("John");
user.sayHi();
```

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}
```



```
constructor(name) {
  this.name = name;
}
```

Class properties versus methods

- Class declaration creates getters, setters, methods in the prototype.
  - They are accessible by all objects created from this class (constructor)
  - Properties are created as properties of the object when a new object is created

```
class User {
   constructor(name ) {  this.name = name;    }
   sayHi() {  console.log(`Hello, ${this.name}!`);
 }

// class is a function
console.log(typeof User); // function

// the prototype will have a reference to the constructor function
console.log(User === User.prototype.constructor); // true

// The methods are in User.prototype, e.g:
console.log(User.prototype.sayHi); // the code of the sayHi method

// there are exactly two methods in the prototype in this example
console.log(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi
```

Inherit from Animal by specifying "extends" Animal

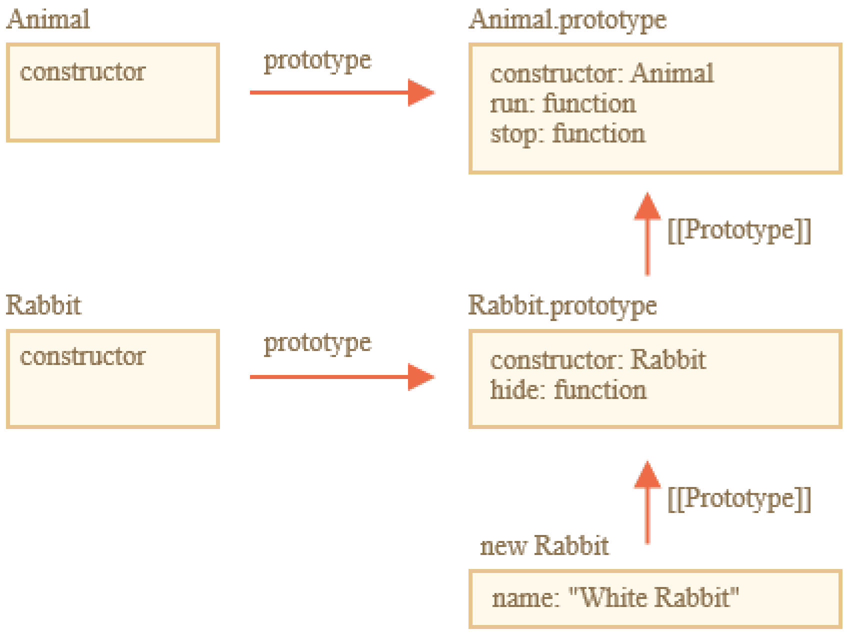
```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {    alert(`${this.name} hides!`);  }}
```

```
class Rabbit extends Animal {
  hide() {  alert(`${this.name} hides!`);  }}
```

```
let rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

- Rabbit code shorter
  - inherits run and stop and constructor

- adds [[Prototype]] reference from Rabbit.prototype to Animal.prototype:
  - if method not found in Rabbit.prototype
    - get from Animal.prototype



Overriding a method

- specify our own stop in Rabbit, it will be used instead

- often don’t want to totally replace a parent method, but build on it
  - do something in our method,
  - call the parent method before/after it or in the process.

- Classes provide "super" keyword for that.
  - super.method(...) to call a parent method.
  - super(...) to call a parent constructor (inside our constructor only)

Overriding a method with super

- Rabbit has the stop method that calls the parent super.stop() in the process.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }}
```

```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }}
```

Overriding constructor with super

- Till now, Rabbit did not have its own constructor.
- if a class extends another class and has no constructor, then an“empty” constructor is generated

```
class Rabbit extends Animal {
  // generated for extending classes without own constructors
  constructor(...args) {
    super(...args);  }}
```

- add a custom constructor to Rabbit. It will specify the earLength in addition to name
  - needs to call super() before using this
  - When a normal constructor runs, it creates an empty object and assigns it to this.
  - when a derived constructor runs it expects parent constructor to do this job.

```
class Rabbit extends Animal {
  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }
```

Encapsulating a property

```
class CoffeeMachine {
  constructor(){   this._waterAmount = 0; }

  setWaterAmount(value) {
    if (value < 0) throw new Error("Negative water");
    this._waterAmount = value;
  }
  getWaterAmount() {
    return this._waterAmount;
  }
}

// create the coffee machine and add water
let coffeeMachine = new CoffeeMachine();
coffeeMachine.setWaterAmount(100);
```