Java Script - Advanced

Sridhar Alagar

Rest parameters

Gather all the remaining arguments passed to a function into an array

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;
  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6

function showName(firstName, lastName, ...titles) {…}
```

Why Rest param when arguments var is there?

```
function sumAll() {
  let sum = 0;
  for (let arg of arguments) sum += arg;
  return sum;
}

alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

arguments is array-like but not an array object.

So, Array methods not available

Also, arguments will always contain all arguments passed

Spread syntax

Does the reverse of Rest parameter

Expands the iterable object into list of elements

```
let arr = [3, 5, 1];

Math.max(...arr); // 5 (spread turns array into a list of arguments)

let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

Rest and Spread – use pattern

Rest parameters are used to create functions that accept any number of arguments

Spread syntax is used to pass an array to functions that normally require a list of many arguments

Class

Introduced in modern JavaScript

```
class User {

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

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

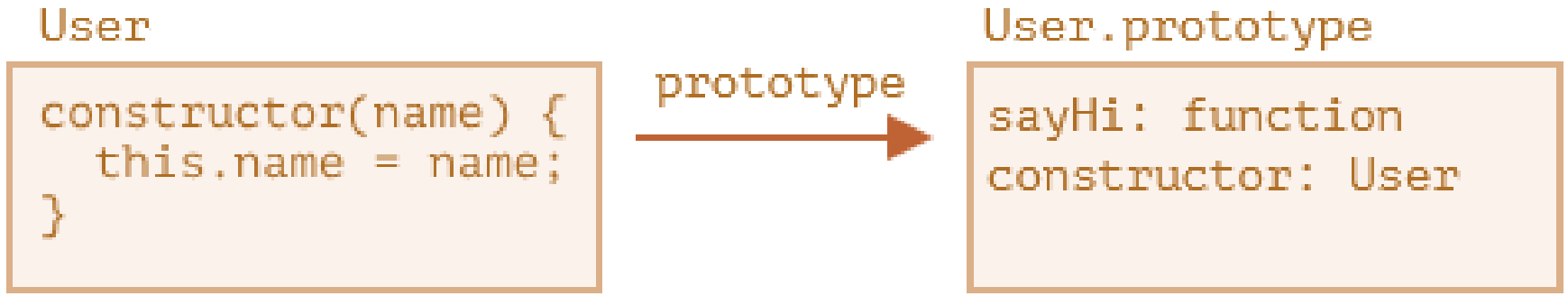
Class

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name);  }
}
// Usage:
let user = new User("John");

//Class is a function
alert(typeof User); // function
```

How Class constructor works?

When class User{…} is declared:

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



Class

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

// ...or, more precisely, the constructor method
alert(User === User.prototype.constructor); // true

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

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

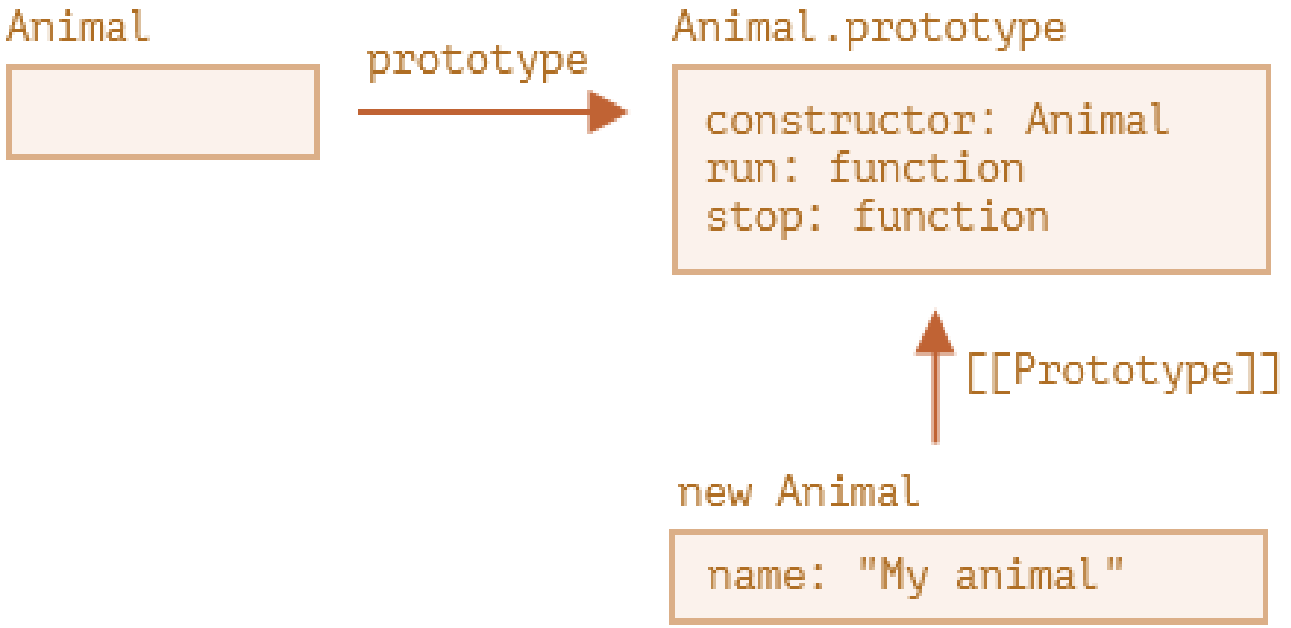
Class – getters and setters

```
class User {

  constructor(name) {
    // invokes the setter
    this.name = name;
  }
  get name() {    return this._name;  }

  set name(value) {
    if (value.length < 4) { alert("Name is too short."); return;  }
    this._name = value;
  }
}
let user = new User("John");
alert(user.name); // John
user = new User(""); // Name is too short.
```

user.name = "Veda"

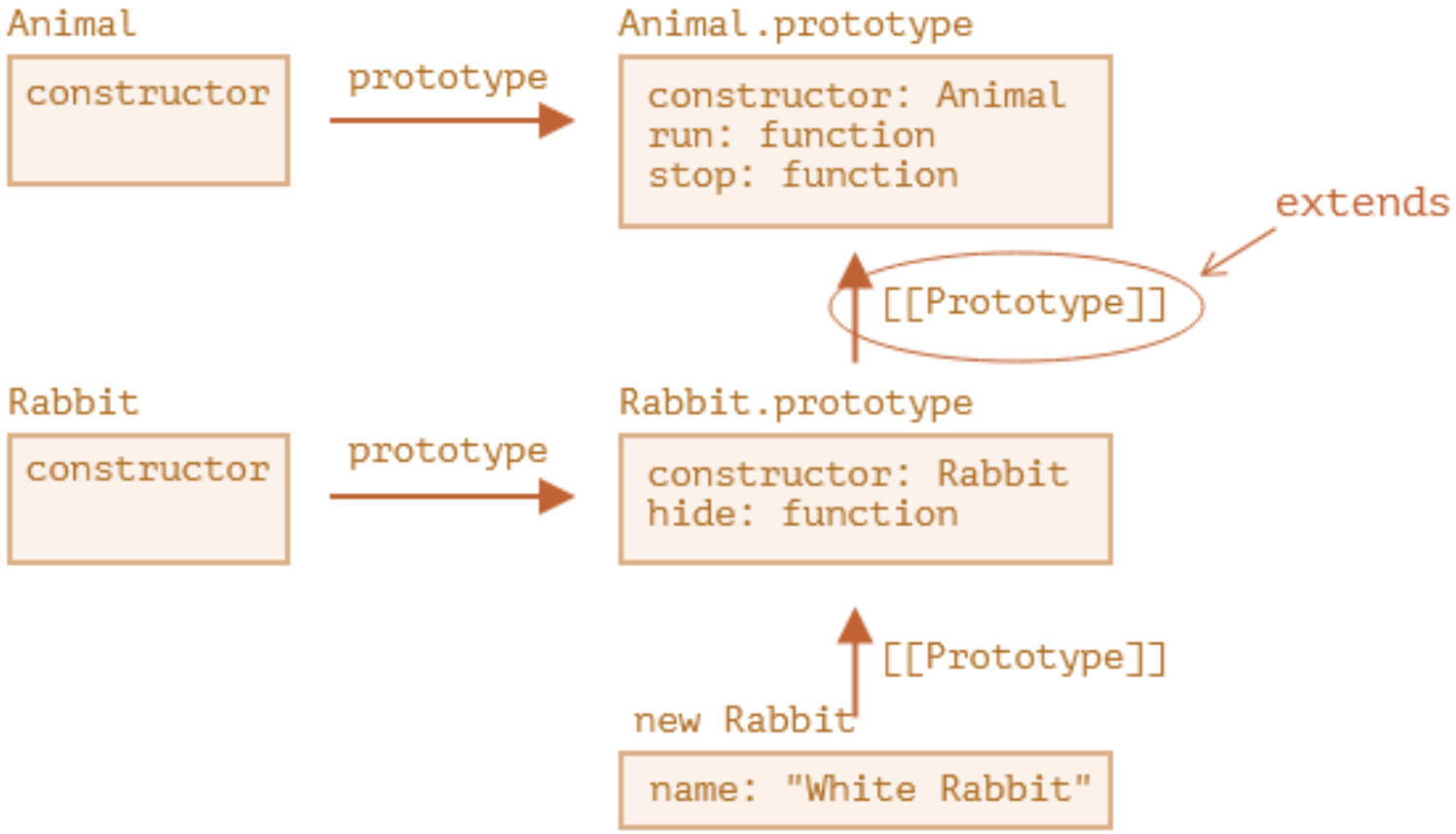
Class Inheritance

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
  }
  stop() {
  }
}

let animal = new Animal("My animal");
```



Class Inheritance

```
class Rabbit extends Animal {
  hide() {}
}
let rabbit = new Rabbit("White Rabbit");
```



Loosing ‘this’

```
class Button {
  constructor(value) {
    this.value = value;
  }
  click() {    alert(this.value);  }
}
let button = new Button("hello");

setTimeout(button.click, 1000); // undefined
```

method is passed around and called in another context, this won’t be a reference to its object anymore.

Use arrow function to avoid losing ‘this’

```
class Button {
  constructor(value) {
    this.value = value;
  }
  click = () => {    alert(this.value);  }
}
let button = new Button("hello");

setTimeout(button.click, 1000); // hello
```

Binding function

```
class Button {
  constructor(value) {  this.value = value;  }
  click() {    alert(this.value);  }
}
let button = new Button("hello");

boundClick = button.click.bind(button)
//this bounded
setTimeout(boundClick, 1000);
```

Binding function

```
class Button {
  constructor(value) {  this.value = value;  }
  click() {    alert(this.value);  }
}
let button = new Button("hello");

setTimeout(button.click.call(button), 1000);

function click (str) { alert(this.value + str)}

click.call(button, “Maya”); // “HelloMaya”
```

function constructor, aka, constructor

Used to create many similar objects

Technically, they are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with "new" operator.

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

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

What does a constructor do?

`new User(...)` does something like:

```
function User(name) {
  // this = {};  (implicitly creates empty object)

  // add properties to this
  this.name = name;
  this.isAdmin = false;

  // return this;  (implicitly)
}
```

Methods in constructor

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

  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}
let john = new User("John");
john.sayHi(); // My name is: John
/*
john = {
   name: "John",
   sayHi: function() { ... }
}
*/
```

Constructor can return only objects

```
function BigUser() {
  this.name = "John";
  return { name: "Godzilla" };  // <-- returns this object
}

alert( new BigUser().name );  // Godzilla

function SmallUser() {
  this.name = "John";

  return 5; // <-- returns this not 5
}

alert( new SmallUser().name );  // John
```

JSON – JavaScript Object Notation

Used for data exchange

Human readable formatted text

```
JSON.stringify(obj) //converts object to JSON
JSON.parse() //converts JSON back to object
```

Different format than Object literal

- Key must be within “ ”
- Single quote not allowed for strings