CS472 WAP
JavaScript Classes

Maharishi International University - Fairfield, Iowa

Wholeness: JavaScript classes are a new syntax that simplifies and enforces proper use of function constructors and prototypal inheritance for creating objects and using good object-oriented design principles such as encapsulation.

Science of Consciousness: Classes are an evolution of JavaScript as a language to support the requirement for good object-oriented design principles needed to develop large applications found in modern web applications. The nature of life is to grow and evolve through layers of abstraction.

Classes

➢ often need to create many objects of the same kind

  ➢ object literal, constructor function

➢ in modern JavaScript,

  ➢ more advanced "class" construct,

  ➢ introduces new features useful for object-oriented programming

➢ look like classes from Java and C#, but are different

  ➢ Based on prototype inheritance and function constructors

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()`.

6

JavaScript classes are (constructor) functions

```
class User {
    constructor(name) { this.name = name; }
    sayHi() { alert(this.name); }
}
// proof: User is a function
alert(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`
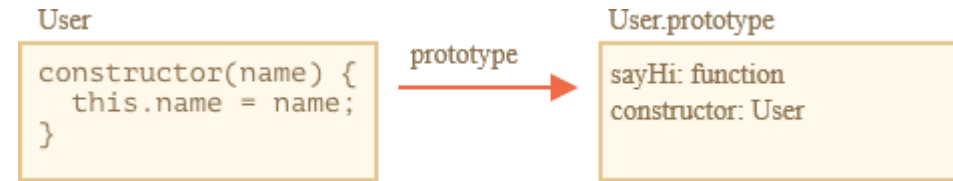 ➢ object has access to class methods.
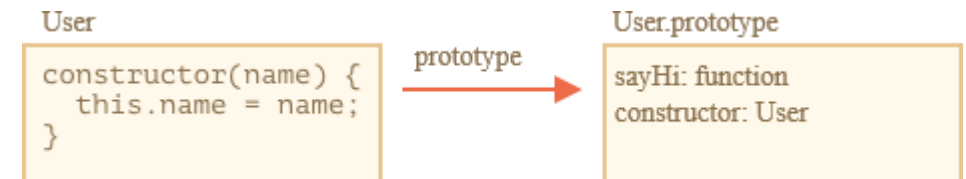
Could write using just constructor function

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

// Add the method to prototype
User.prototype.sayHi = function ()
{
    alert(this.name);
};

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

Could write using just constructor function

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

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

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

Differences and advantages of ES6 class syntax

➤ function created by class is labelled by a special internal property `[[FunctionKind]]:"classConstructor"`.

➤ Error message if forget to use '`new`'
```
class User {
    constructor() { }
}
alert(typeof User); // function
User(); // Error: Class constructor User cannot be invoked without 'new'
```

➤ string representation of a class constructor in most JavaScript engines starts with the "class…"
```
class User {
    constructor() { }
}
alert(User); // class User { ... }
```

➤ Class methods are non-enumerable. sets enumerable flag to false for all methods in "prototype".
  ➤ good, because if we for..in over object, usually don't want class methods

➤ Classes always `use strict`
➤ …

Getters and setters

➢ Traditional class-oriented languages typically have methods called getters and setters for accessing state information in properties
  ➢ Often contain small amounts of code for constraint checking

```javascript
class User {
    #privatename;
    constructor(name) {
        this.name = name;   // calls the setter
    }
    get name() {
        return this.#privatename;    //property must match the name used in the setter
    }
    set name(value) {
        if (value.length < 4) {
            alert("Name is too short.");
        } else {
            this.#privatename = value;    //must set a property name different from the setter name
        }
    }
}
let user = new User("John");
console.log(user.name);   // calling the getter
user.name = 'Fred';   // calling the setter
user2 = new User("");   // Name too short.
```

11

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

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

fred = new User();
console.log(fred);
console.log(fred.__proto__);
console.log(User.prototype);
fred.sayHi();
bob = new User("Bob");
console.log(bob);
bob.sayHi();
```

Class inheritance

```
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 {

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

    hide() {
        alert(`${this.name} hides!`);
    }
}
```
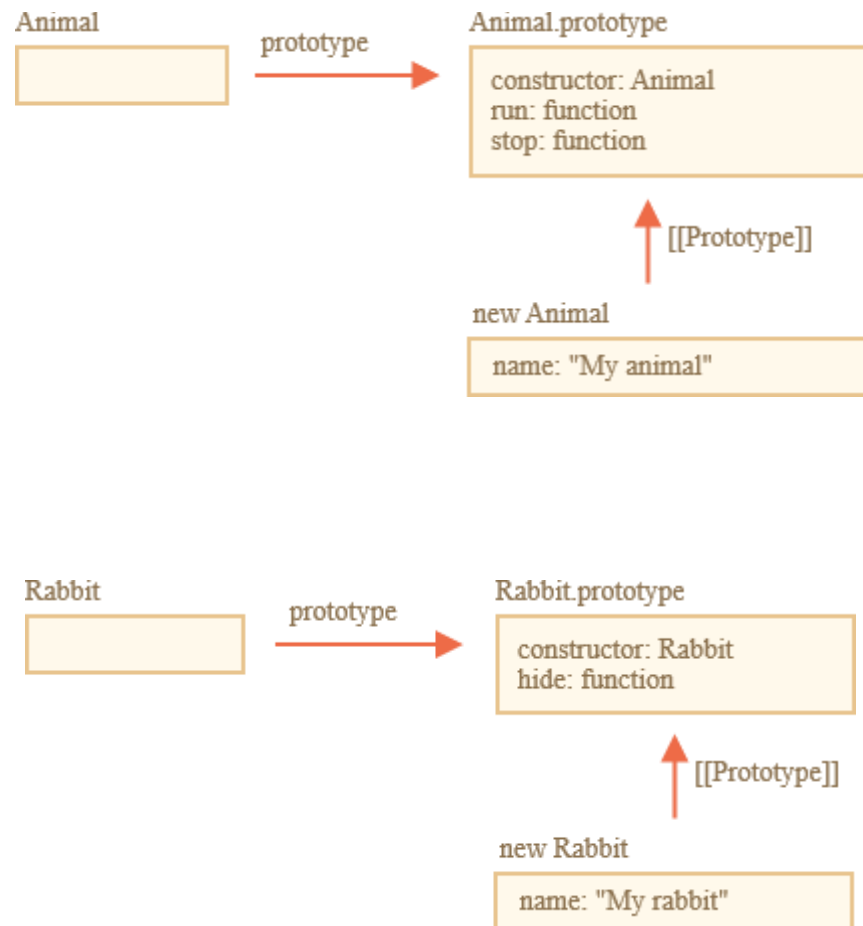
Inherit from `Animal` by specifying "extends" Animal

```
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`
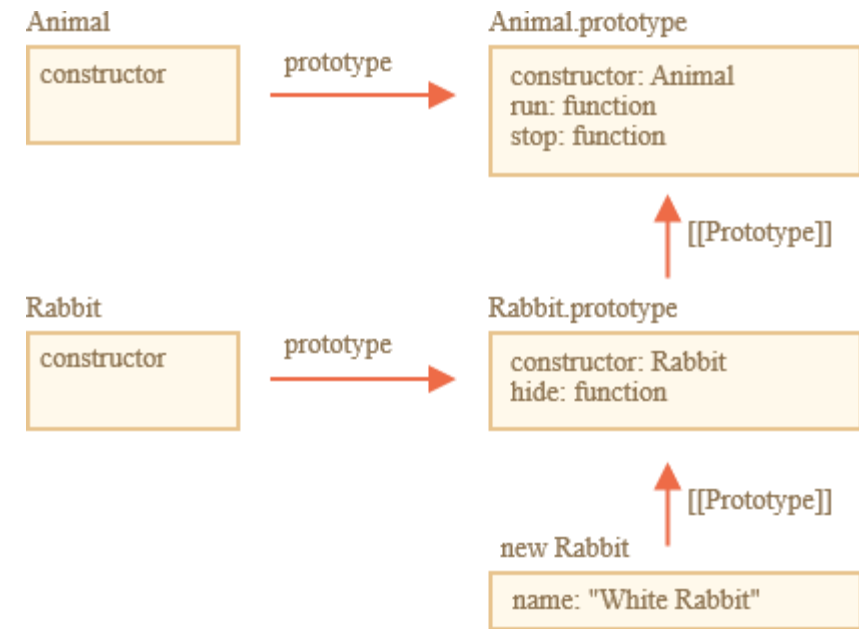
Overriding a method

➤ 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;
    }
}
```

Static properties and methods

- can assign a method to the class function itself, not to its "prototype".
  - Such methods called static

- value of this in `User.staticMethod()` call is the class constructor `User` itself
  - "object before dot" rule

- static methods used for functions that belong to the class
  - not to any particular object

```javascript
class User {
    static staticMethod() {
        alert(this === User);
    }
}
User.staticMethod(); // true
```

```javascript
//same as assigning it as a property directly

class User {

}

User.staticMethod = function() {
    alert(this === User);
};
```

Static methods

➢ `Article.compare` is a means to compare articles
  ➢ not a method of an article, but rather of the whole class.

```javascript
class Article {
    constructor(title, date) {
        this.title = title;
        this.date = date;
    }
    static compare(articleA, articleB) {
        return articleA.date - articleB.date;
    }
}
// usage
let articles = [
    new Article("HTML", new Date(2019, 1, 1)),
    new Article("CSS", new Date(2019, 0, 1)),
    new Article("JavaScript", new Date(2019, 11, 1))
];
articles.sort(Article.compare);
alert(articles[0].title); // CSS
```

Static properties

➢ Static properties are also possible,
   ➢ look like regular class properties, but prepended by `static`:

```
class Article {
    static publisher = "Ilya Kantor";
}

alert(Article.publisher); // Ilya Kantor
// That is the same as a direct assignment to Article:
Article.publisher = "Ilya Kantor";
```

Inheritance of static methods

```javascript
class Animal {
    constructor(name, speed) {
        this.speed = speed;
        this.name = name;
    }
    run(speed = 0) {
        this.speed += speed;
        alert(`${this.name} runs with speed ${this.speed}.`);
    }
    static compare(animalA, animalB) {
        return animalA.speed - animalB.speed;
    }
}
// Inherit from Animal
class Rabbit extends Animal {
    hide() {
        alert(`${this.name} hides!`);
    }
}
let rabbits = [
    new Rabbit("White Rabbit", 10),
    new Rabbit("Black Rabbit", 5)
];
rabbits.sort(Rabbit.compare);
rabbits[0].run(); // Black Rabbit runs with speed 5.
```
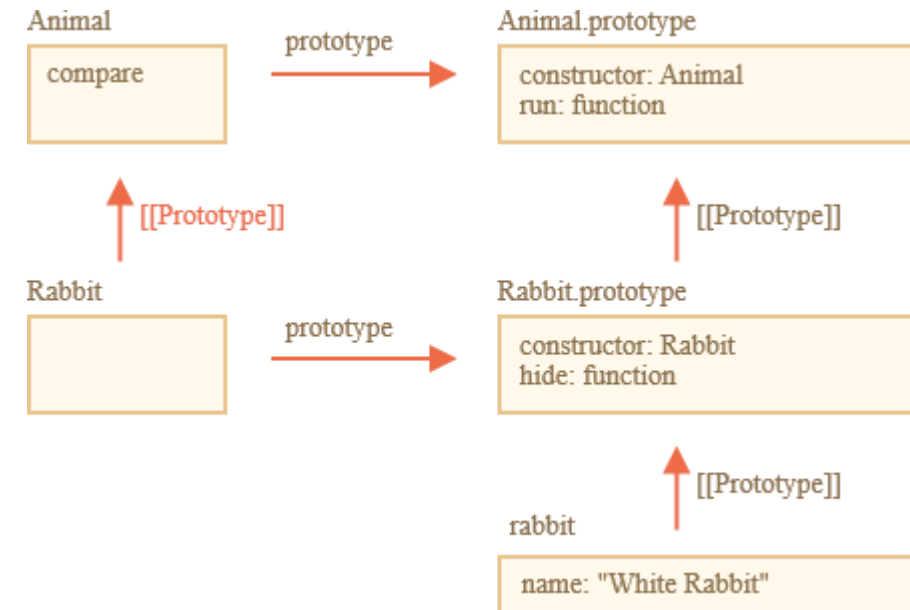

➢ `Rabbit extends Animal` **creates two** `[[Prototype]]` references
  ➢ `Rabbit` **constructor prototypally inherits from** `Animal` constructor
  ➢ `Rabbit.prototype` **prototypally inherits from** `Animal.prototype`.

➢ Static methods belong to class "as a whole",
  ➢ Not accessible to specific object instances as object methods
  ➢ i.e., `Rabbit.compare`, **not** `longEar.compare`

21

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Evolution through Layers of Abstraction

1. JavaScript classes are a helpful syntax that abstracts out details of function constructors and prototypal inheritance for creating objects.

2. The extends and super keywords cause objects and properties to be set in the function prototype and [[Prototype]] properties.

_____

3. **Transcendental consciousness**. Is the simplest state of awareness. It abstracts away everything and is also the basis of everything.

4. **Impulses within the transcendental field:** Impulses at this level are the finest layer of existence and represent the first abstraction of knower, known, and process of knowing when consciousness is aware of itself.

5. **Wholeness moving within itself:** In unity consciousness one appreciates all layers of existence as expressions and abstractions over pure consciousness, the unified field of existence.